

O B M

A MACHINE INDEPENDENT MACHINE
ORIENTED LANGUAGE

GÖRAN FRIES

CODEN: LUNFD6/(NFIP-3005)/1-83/(1978)

1978



LUND UNIVERSITY
Department of Computer Sciences



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41553421_0030

ABSTRACT	1
1. INTRODUCTION	2
1.1 PURPOSE OF OBM	2
1.2 THE STRUCTURE OF THE OBM SYSTEM	3
1.3 NOTATION FOR EXTRACT	4
O B M	
2. A MACHINE INDEPENDENT MACHINE ORIENTED LANGUAGE	5
2.1 GENERAL DESIGN	5
2.2 PRIMITIVE SYNTAX	8
GÖRAN FRIES	
CODEN: LUNFD6/(NFIP-3005)/1-83/(1978)	
3. THE EXECUTABLE SEGMENT	10
3.1 THE EXECUTABLE SEGMENT	10
3.2 PSEUDOPERATORS	12
4. THE DATA DEFINITION AND MEMORY ALLOCATION OF A SEGMENT	13
4.1 TYPE DEFINITIONS	13
4.1.1 THE TYPE	13
4.1.2 THE ALLOCATION PRINCIPLE	20
4.2 MEMORY ALLOCATION	24
4.3 VARIABLES AND CONSTANTS	27
4.4 THE EQUIVALENCE STATEMENT	29
4.5 EXAM	30
This work has been supported by The Swedish National Board for Technical Development (75-3354, 76-3609, 77-4426).	
5. INSTRUCTIONS	36
5.1 GENERAL CONVENTIONS AND CONCEPTS	36
5.1.1 ALLOCATION	36
5.1.2 SCOPE OF DEFINITIONS AND OTHER STRUCTURE	38
5.1.3 PRIMITIVE	40
5.2 INSTRUCTION PART OF SEGMENT	42
5.3 ARITHMETICAL AND LOGICAL INSTRUCTIONS	43
5.3.1 ARITHMETICAL	43
5.3.2 LOGICAL	45

LUND, OCTOBER 19

LUND UNIVERSITY
INSTITUTE OF COMPUTER SCIENCES
SÖLVEGATAN 11 A
S-223 62 LUND, SWEDEN

U. N.
A MACHINE INDEPENDENT MACHINE
ORIENTED LANGUAGE
LÖNN PRIES
CODING UNIT (U. N. - SÖLVEGATAN 11 A)



This work has been supported by the Swedish National Board
for Technical Development (72-3354, 75-400, 77-400)

C O N T E N T S

ABSTRACT	1
1. INTRODUCTION	2
1.1 PURPOSE OF OBM	2
1.2 THE STRUCTURE OF THE OBM SYSTEM	2
1.3 NOTATION FOR SYNTACTIX DESCRIPTION	4
2. THE OBM PROGRAM	5
2.1 GENERAL DESIGN OF OBM	5
2.2 PRIMITIVE SYNTACTIC AND SEMANTIC UNITS	8
3. THE EXECUTABLE OBM SEGMENT	10
3.1 THE EXECUTABLE SEGMENT	10
3.2 PSEUDOOPERATORS	12
4. THE DATA DEFINITION AND MEMORY ALLOCATION OF A SEGMENT	13
4.1 TYPE DEFINITIONS	13
4.1.1 THE TYPE	13
4.1.2 THE ALPHASTATEMENT	20
4.2 MEMORY ALLOCATION	24
4.3 VARIABLES AND CONSTANTS	27
4.4 THE EQUIVALENCE STATEMENT	29
4.5 EXAMPLE OF DATA DEFINITIONS	30
5. INSTRUCTIONS	36
5.1. GENERAL CONVENTIONS AND CONCEPTS	36
5.1.1 ADDRESSING	36
5.1.2 SCOPE OF DEFINITIONS AND BLOCK STRUCTURE ...	38
5.1.3 TRIMMING	40
5.2 INSTRUCTION PART OF SEGMENT	41
5.3 ARITHMETICAL AND LOGICAL INSTRUCTIONS	43
5.3.1 ARITHMETIC	43
5.3.2 LOGIC	45

C O N T E N T S

1	ABSTRACT	
2	INTRODUCTION	1
3	PURPOSE OF GBM	1.1
4	THE STRUCTURE OF THE GBM SYSTEM	1.2
5	NOTATION FOR SYNTACTIC DESCRIPTION	1.3
6	THE GBM PROGRAM	2
7	GENERAL DESIGN OF GBM	2.1
8	PRIMITIVE SYNTACTIC AND SEMANTIC UNITS	2.2
9	THE EXECUTABLE GBM SEGMENT	3
10	THE EXECUTABLE SEGMENT	3.1
11	PSEUDOOPERATORS	3.2
12	THE DATA DEFINITION AND MEMORY ALLOCATION OF A SEGMENT	4
13	TYPE DEFINITIONS	4.1
14	THE TYPE	4.1.1
15	THE ALPHABET	4.1.2
16	MEMORY ALLOCATION	4.2
17	VARIABLES AND CONSTANTS	4.3
18	THE EQUIVALENCE STATEMENT	4.4
19	EXAMPLE OF DATA DEFINITIONS	4.5
20	INSTRUCTIONS	5
21	GENERAL CONVENTIONS AND CONCEPTS	5.1
22	ADDRESSING	5.1.1
23	SCOPES OF DEFINITIONS AND ERROR HANDLING	5.1.2
24	TRIMMING	5.1.3
25	INSTRUCTION PART OF SEGMENT	5.2
26	ARITHMETICAL AND LOGICAL INSTRUCTIONS	5.3
27	ARITHMETIC	5.3.1
28	LOGIC	5.3.2

5.4	BRANCHING	46
5.4.1	BRANCHING IN OBM	46
5.4.2	CASE AND LABEL ASSIGNMENT	47
5.4.3	UNCONDITIONAL BRANCH STATEMENTS	47
5.4.4	TEST OR CONDITIONAL BRANCH STATEMENTS.....	49
6.	HIGHER LEVELLED INSTRUCTIONS	52
6.1	STRING AND STACK HANDLING	52
6.1.1	STRING HANDLING	52
6.1.2	STACK HANDLING	55
6.2	LOOPS	56
6.3	SUBROUTINES	57
6.4	SEARCH INSTRUCTIONS	62
7.	OPERATING SYSTEM COMMUNICATION INSTRUCTIONS	65
7.1	SYSTEM INSTRUCTIONS	65
7.2	PROGRAMS AND ACTIVITIES	66
7.2.1	EXECUTION PRIMITIVES	67
7.2.2	SYNCHRONIZATION	69
7.2.3	ABSOLUTE PROGRAM LINKING	70
7.3	SECONDARY MEMORY CONTROL	71
7.3.1	I/O OPERATIONS	72
7.3.2	I/O DEVICE CONTROL	74
8.	INTERRUPTS AND INTERRUPT SEGMENTS	77
8.1	ERROR INTERRUPTS	77
8.2	I/O INTERRUPTS	78
8.3	INTERRUPTS NAMES	80
9.	PROGRAM DESCRIPTION SEGMENT	81
	APPENDIX A	10 SIDOR
	APPENDIX B	1 SIDA
	APPENDIX C	1 SIDA

ABSTRACT

This report is a definition and description of a machine oriented language OBM. This language is intended to be a machine independent language. The purpose of OBM is to use it in a system aiming at compatibility and portability. Within the language it is possible to demand the "Computer Architecture" and this must be simulated, if not available in hardware. OBM may be regarded as an intermediate language between high level languages and machine code, on the other hand OBM may be regarded as a fictive, flexible computer.

1. INTRODUCTION

1.1 PURPOSE OF OBM

OBM is a machine independent machine-oriented language. It is developed to be the basic language for system software, designed to give compatible implementations on different computers [1]. OBM should be a good assembly language for most existing computers, and it should be suitable for representing structures from programs written in any high level language. This is further discussed in [2].

A system where output data is dependent only on input and program, and independent of computer is wanted. A source deck as input to a computer shall give the same result on any computer. For efficiency reasons it is possible to specify that the result may differ in some respects, but still the source deck is suitable as input to any computer. This is portability. It shall be possible to specify the amount of compatibility desired. In what respects the full compatibility may be sacrificed must be perfectly well defined within the OBM program. The system and the compatibility does also include commands to the operating system: A discussion of why compatibility and portability is needed is given in [3].

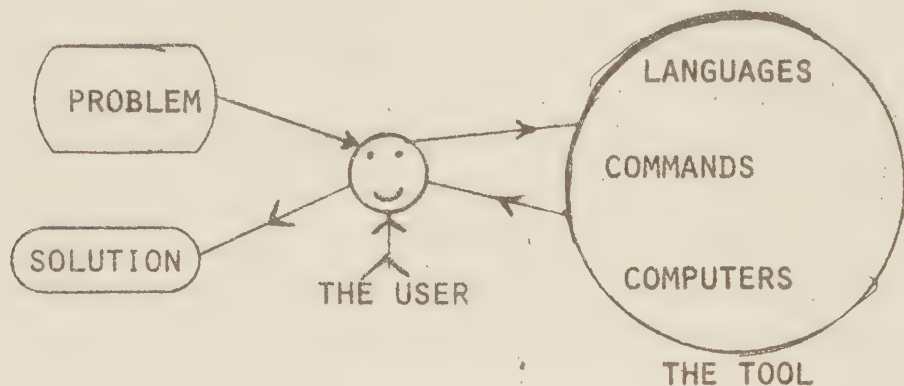
1.2 THE STRUCTURE OF THE OBM SYSTEM

In order to solve the compatibility problem, a system is constructed consisting of high level languages and compilers from these to an intermediate language OBM, a machine-oriented language OBM, a number of computers and translators from OBM to these. The system is also embedded in an operating system OBMOS. All high level languages in the system may be regarded as implemented on the fictive OBM machine. This is a flexible machine with a great number of word lengths and word dispositions to be chosen by the programmer. This fictive machine is then simulated by a real (host) machine in the system. In order to control efficiency, the programmer can choose the word-length and disposition in terms of the actual host computer, and then of course sacrifice the full compatibility.

1.2

It should be pointed out that our system is not a system that handles just a program or a number of programs. The natural unit is the job. A job may be considered as a solution to a problem and given to a computer system. The job includes system commands and a number of programs, and sometimes data.

We may consider the system to include also the programming language, the system is a tool that may be used to solve problems. The problem is transformed by the user to a job, expressed in terms of the system, and given to the system to be processed.



A high level language is today not completely defined until it is implemented. The word length, arithmetic, overflow action and such features are defined by the implementer in terms of the machine on which the language is implemented. This will lead to a machine dependent language, also for a "standardized" programming language. In the OBM system there is only one implementation, on the fictive OBM computer, and the standard is given once. But the OBM computer has e.g. several word lengths. Shall only one of them be selected by the implementer, and then impossible to change. No, the implementer shall only define a standard, and then a method for the programmer to select his own "standard" if he wants to. This selection is carried out by means of parameters to the compiler.

The heart of this system, the machine-oriented language OBM, will be described and defined later in this report. The other parts of the system will be described in other later reports.

1.2

In a compatible system we will have:

- 1) Machine independent compilers.
 - 2) An operating system, OBMOS.
 - 3) A command language, OBMOS-CL.
 - 4) A machine independent assembly language, OBM.
 - 5) A number of different computers.
 - 6) One OBM translator for each computer.
 - 7) One OBMOS runtime system for each computer.
- This is also suitable for a computer network.

1.3

'NOTATION FOR SYNTACTIC DESCRIPTION

The syntactic description will be given in a modified Bachus-Naur Form (BNF). All non-terminal symbols are strings of lower-case letters, all other symbols, including upper-case letters, are terminals. Some special symbols are used for the description and shall not be included in the OBM language. The following modifications to BNF are used:

- | | | | |
|-------------|---------------------------|---|--|
| { } | * | repeat the group in { } | zero or more times |
| unit | * | repeat the <u>unit</u> | zero or more times |
| { } | + | repeat the group at least once | |
| unit | + | repeat unit at least once | |
| { } | _n ^m | repeat at least n times and | |
| unit | _n ^m | at most m times, if n omitted | zero is assumed. |
| { } | | if group does not contain alternatives, | the group may be omitted or used once. |
| { ... } | | if group contains alternatives, | one alternative should be choosen. |

E X A M P L E :

id → letter{letter|digit}*
means that id is defined as one letter followed by any number of letters or digits. A2BC is valid id.

The non-terminals are choosen to give some help with the semantic definitions. Therefore some different non-terminal symbols are defined to have the same syntactic meaning, but they have different "names" because of their different semantic use.

2. THE OBM PROGRAM

2.1 GENERAL DESIGN OF OBM

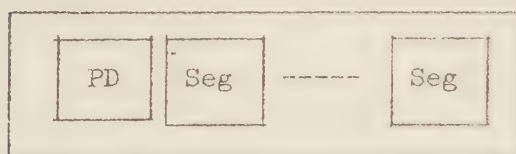
An OBM source file is a sequence of symbolic OBM segments. There are segments of three different types; Program description segment, interrupt segment and executable segment.

The most important is the executable segment. This segment contains the normal code. An interrupt segment may be used to specify interrupt routines other than the default routines in the system.

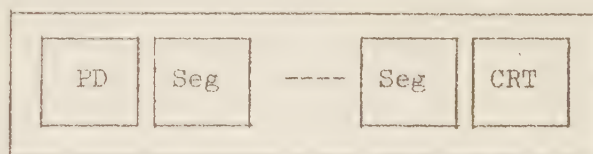
A program description segment may be used to pass information about the program structure to the linker routine of the operating system. The program description segment gives the programmer a means to pass his knowledge about the program structure to the linker, this may also be done through an operating system command. The linker may use this information together with its own knowledge of the actual computer to build an appropriate segmented program.

An OBM source file is often an OBM program, but it may also be just a sequence of segments intended for library use. The source file may be used as a program if it contains at least one segment with a start address. A program description is not a necessary part of an OBM program. More than one program description segment in a source file is not permitted.

An OBM source file is translated to an OBM relocatable file. The relocatable file will be a sequence of relocatable segments, and a cross reference table.



Program File with Program Description



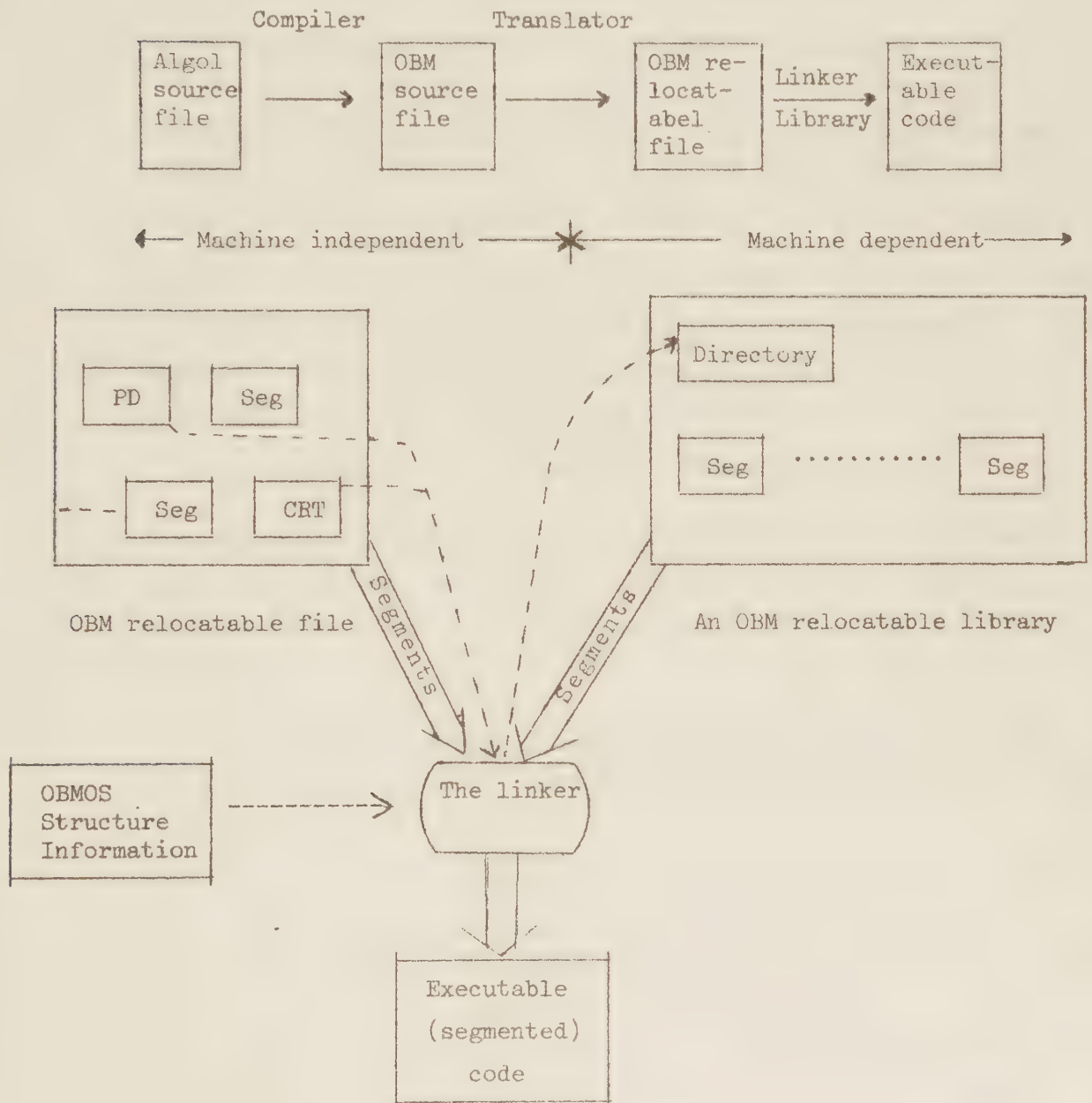
Relocatable File with Cross Reference Table

The design of a program file is machine independent. It is a symbolic string and its syntax is defined as a part of the OBM definition. The relocatable file contains segment in a relocatable machine code for some computers, and a table of cross references between segments in the file.

2.1

The details here are implementation dependent, and the file may be organized with labels and pointers in an appropriate way, but logically it must be a file as described above.

The operating system should also provide means for including a relocatable file without program description in a library. The most common OBM programmer is a compiler and the following figure will give a typical example of running a simple Algol program.



Linking of an OBM program

2.1

The linker will produce an executable program from the segments in the OBM relocatable file and necessary segments from the library. Depending on the size of the host computer the executable code will be segmented or not. The linker determines the segmentation using information from the program description, cross reference tables and information coming from the OBMOS commands. The program description and the OBMOS structure information are no demands only wishes. It has to be like this because the programmer who writes this information do not necessarily know anything about the host computer.

This chapter has only given a brief logical description of the OBM program and the use of such a program. The rest of the report will give the definitions of the language, starting with the executable segment in chapter 3. The interrupt segment is defined in chapter 8, and the program description and the source file structure in chapter 9.

The source file is a symbolic string according to the syntax of OBM. The logical structure of the file is that it is a sequence of segments, and the segments are sequences of statements.

A statement has the following general structure:

```
label:operator,specification    parameterfield,---- ;
```

A statement is characterized by the operator field, and the design and presence of the other fields are dependent on the operator. All statements contains at least an operator and the statement delimiter;.

The segment is also important for the scope of definitions. I will define some words used to express scope properties, and used in the rest of the report.

- i/. Local - means that its scope is a part of a segment.
- ii/ Global - means that its scope is a segment.
- iii/ External- means that it is defined outside the segment where it is used as external.
- iv/ Externally referencable - means that it is global within the segment, but its scope may be external to other segments.

2.2

PRIMITIVE SYNTACTIC AND SEMANTIC UNITS

The character set used in OBM statements (except string-data), is a subset of the international standard character set ISO 646.

The following characters are included in the subset:

1/ Digits:	0 1 2 3 4 5 6 7 8 9
2/ Letters:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
3/ Special characters:	() * + - / . , < = > : ; ' space

The space will be written as ϵ in this report!

Syntax:

digit	$\rightarrow$	0 nonzero
nonzero	$\rightarrow$	1 2 3 4 5 6 7 8 9
letter	$\rightarrow$	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
id	$\rightarrow$	letter{letter digit} [*]
integer	$\rightarrow$	0 nonzero{digit} [*]
del	$\rightarrow$	ϵ ;

Notice that the length of an integer is arbitrary, but the different OBM statements have different restrictions on integer size. Also an identifier has an arbitrary length, but only six characters are significant. Some operators consists of more than six characters, but also for operators only the six first characters are significant. As a statement delimiter del is used.

Syntax:

adr	$\rightarrow$	id id(ind) id(ind,ind)
ind	$\rightarrow$	id integer - integer
data	$\rightarrow$	id
label	$\rightarrow$	id
trlab	$\rightarrow$	id
lab	$\rightarrow$	id
type	$\rightarrow$	id

2.2

Semantics

adr	is used to address a data word. The identifier, <u>id</u> , used must have been defined by a DATA statement, that means it must have appeared as a <u>label</u> on a DATA statement.
lab	is an identifier that has been defined by an executable instruction statement (appeared as a <u>label</u> on such an instruction).
trlab	The same as lab, but it must be possible to jump to the defining instruction ("transfer label").

As we will see later there are labels on statement that are executable, but it is impossible to jump to those statements. Such statements are used for a kind of dynamical data definitions like switches.

ind	This is used as an index. The <u>id</u> must be defined as a data word, as for <u>adr</u> . The <u>integer</u> must be less than or equal to $2^{16}-1$. There are also some restrictions on the word length if <u>id</u> is used. Such a data word must have a length <64 bits, and is regarded as an integer. Decimal arithmetic is not possible for indices.
data	is an identifier that has been defined by a DATA statement.
type	is a name of a type defined by the TYPE statement.

The syntax of an OBM program is as follows:

Syntax

obmprog $\rightarrow$ {progdescr} {segment|irptseg}⁺ PROGEND del

Semantics

obmprog.	is a number of segments given to the translator. Segments are translated separately but some cross references between segments, are produced.
progdescr	is described in 9.
segment	is the most important part of a program. The main part of this report describes this segment. The description is in 3 - 7.
irptseg	is a segment where user interrupt service is defined. This segment is described in 8.
PROGEND	is a statement used to physically finish the input text to the translator.

3. THE EXECUTABLE O B M SEGMENT

3.1 THE EXECUTABLE SEGMENT

A segment is a unit where all labels are local and all references are local if not otherwise defined. A segment is separately translated into relocatable code (machine dependent). A sequence of several segments is the input to the OBM translator, and the result will be a file of relocatable elements and a segment table.

The storage of these sequences are of course computer dependent.

In some respects a segment is just a number of statements translated as a unit, independent of all other such units. But more important is that a segment is a logical unit of statements that will reside in memory together, if permitted by memory size on the host computer.

For large programs or small computers (according to memory), the OBMOS system must use a technique with overlay segments or paging. To the system an OBM segment is a unit used to build segments on the actual computer, only if the size of the OBM-segment is too great, then such a segment may be divided by the OBMOS collector to smaller segments. In order to achieve an efficient actual segmentation, the OBM segment should be a unit with mostly internal references and with a few, non-frequent external references to other segments. To avoid blind division of segments they should not be large, but large enough to avoid too frequent external references.

Syntax:

```

segment      → label:SEGMENT  $\epsilon^+$  {begin} del
               {exlab|exref|exsub|datadescr} *
               {instruction} * END del

begin        → trlab

exlab         → EXDEF  $\epsilon^+$  trlab{,trlab} * del

exref         → EXREF  $\epsilon^+$  id{,id} * del

exsub         → label:SUBREF  $\epsilon^+$  param{,param} * del

datadescr     → typestat|poolstat|datastat|alphastat|
               pseudo|common|stringstat

```


3.1

Semantics:

SEGMENT

The purpose of the first statement in a segment is

- i) To give a signal to the translator that a new segment is to be translated.
- ii) To define label as the name of the segment.
- iii) If begin is given, to bind a start address to the segment. This makes it possible for OBMOS to use the segment as the main segment for a program. The begin address must be given as a transfer label within the segment.

exlab

This statement defines trlab to be accessible from other segments as transfer labels.

exref

Here the identifiers id are defined as transfer labels, that are external. These id are transfer labels, but they do not appear as labels on any statement within the segment. During the linking the OBMOS collector must find them as defined external transfer labels in some other included segment.

exsub

Here label is defined as the name of a subroutine with formal parameters given by param. The subroutine body is not included in this segment, and the OBMOS collector must find it in another included segment. Often the routine is defined in a segment in the public library. The syntax and semantics of param is defined in 6.3.

datadescr

A data description statement is used to define machine concepts such as types of data and operators, memory structure and requirements.

typestat, poolstat, datastat, alphastat and common are defined later in chapter 4, pseudo is defined in 3.2.

instruction

See chapter 5!

END

This statement is used to give a signal to the translator to finish the translation of the segment. It is illegal to make an attempt to execute the statement "following" the last statement of the segment. Trying to execute the END statement will cause an end of segment interrupt (SEG, see 8!).

3.2 PSEUDO - OPERATORS

Statements containing pseudo-operators may be present anywhere in the symbolic program. They do not produce any code or datastructure in the object program. They are used to control the translator. They are used to change the mode of translation and they may be switched on or off any number of times.

Syntax

pseudo → checkop|arithmfault|recursion|radix

Semantics

checkop is used for turning on or off the generating of indexcheck code in object program. (See 4.3!)

arithmfault is used to suppress the generation of time consuming arithmetic fault test code in object program. (See 5.3!)

recursion is used to set a limit on recursive subroutine stack length. (See 6.3!)

radix is used to change the radix for integer input to some instruction. Most integer operands will always be interpreted as decimal. (See 4.1!)

E x a m p l e :

```

ADAM:SEGMENT;
      EXDEF  L0,L1,L2;EXREF  L3,L4,L5,L6;
      SUB:SUBREF  (32,1),(32,0);
          .
          .      data part
          .
          .
          .      instruction part
          .
      END;
```

The segment consists of a SEGMENT statement followed by a non-executable data and type definition part, followed by an executable instruction part and ended by the END statement.

4. THE DATA DEFINITION AND MEMORY ALLOCATION OF A SEGMENT

The first part of a segment, after the SEGMENT statement, is a non-executable part used to define some concepts used in the executable part. The executable part starts with the first executable instruction. The OBM machine is a flexible one, and it has a great number of wordlengths and dispositions. The programmer must tell the machine which of those he wants to use. In the data definition part, statements that will define certain modes and give them names may be included. These names (modes, types) are then used both in memory allocation and in the instruction part.

In OBM the memory structure must be defined. This is not just a description of some computer (host computer), but a command telling the computer how to structure its memory. Here we may force the computer to act just as we want it to act. The memory structure may be defined in an extremely flexible way, but care must be taken in this definition, or inefficient programs will be produced.

4.1 TYPE DEFINITIONS

4.1.1 THE TYPE

One of the fundamental concepts in OBM is the type. A type has two logically different purposes. The first purpose is to define a memory structure and the word length of data words.

The second purpose is to define arithmetical and other operators.

As an example the + operator must use a type to define what kind of plus it is. Important properties of an operator is not only the length of its operands, but also if the operand should be interpreted as a real or an integer, the sign representation and how the arithmetic should be performed. A type intended for the second purpose contains enough information to be used also for the first purpose. Therefore OBM does only have one kind of type definition. But it must be noticed even if a "real" type is used in memory definitions a data word has never the property of being real or using truncating arithmetic, having only a word length.

4.1.1

The same data word may be used as an operand to operators of any type. It is permitted to use the same operand for both integer and real number arithmetic, of course the programmer is responsible for eventual logical errors.

There are five different kind of types to be defined:

INTEGER, REAL, STRING, NODE and BIT.

Syntax:

typestat	→	label:TYPE ϵ^+ tpar del {node}
tpar	→	INTEGER,n{,store{,arithm}} REAL,exp,mant,base,sign,trunc REAL,ACTUAL STRING,ch,used,word,alpha NODE,pw,n BIT,n{,store}
n	→	integer
exp	→	integer
munt	→	integer
base	→	2 4 8 16 10
ch	→	integer
used	→	integer
word	→	integer
pw	→	integer
store	→	N W S M E
arithm	→	N S O C D
sign	→	A B C
trunc	→	N C R P
alpha	→	id
node	→	{wordstat} _n ⁿ
wordstat	→	WORD{,E} ϵ^+ length{,length}* del
length	→	integer integer*integer

Semantics

typestat is used to define label as a type. The type is then used for storage allocation and for operator definitions. The type is also used as information during conversions.

4.1.1 Type_properties_used_for_storage_allocation

The type is used by other statements to specify the properties of an OBM word and its storage in the actual computer.

INTEGER defines a word of n bits, stored according to store, see table!
Restrictions: $2 \leq n \leq 255$

store	description	storage in bits	significance in bits
void or			
N	normal	$? \geq n$	n
W	word.(s)	$k * c \geq n$	n
S	Significant word(s)	$k * c \geq n$	$k * c$
M	minimum	$? \geq n$	$? \geq n$
E	exact	n	n

c = actual machine word length
? = dependent on implementation
k = the least integer such that $k * c \geq n$

REAL defines a word equivalent to INTEGER,exp+mant,N;
Restrictions: $4 \leq \text{exp+mant} \leq 255$

STRING defines a word equivalent to BIT,ch*word;
Restrictions: $2 \leq \text{used} \leq \text{ch} \leq 8$
 $\text{word} * \text{ch} \leq 255$
 $\text{word} \geq 1$

BIT defines a string of bits (no signbit), storage equivalent to INTEGER,n,N;
Restrictions: $1 \leq n \leq 255$

NODE defines a structured word(node)consisting of pw partial words (numbered from 1 to pw) and described in n immediately following wordstat. Here a number of partial words, each with given length are defined. If E is used it means that the partial words are stored compacted to a bit string, else the actual length of a partial word is machine dependent ($\geq \text{length}$). If length is of the form integer * integer it means integer partial words each of length integer.

4.1.1 Restrictions: The length of a partial word must be in the closed interval [1,255]. The sum of all partial wordlengths compacted together (E-option) must be ≤ 255 . Only partial words defined in the same WORD,E statement are compacted together.

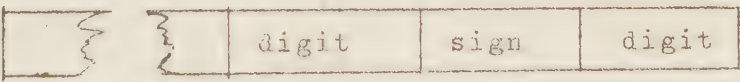
Type_properties_used_for_operator_definitions

Types are also used to specify instructions and define operators.

INTEGER an integer is a fix point number and the sign may be stored in different ways, according to arithm.
Also decimal arithmetic is possible.

arithm	description
void	
N	Sign as actual computer
S	Signbit, no complement
O	Signbit and One's complement
C	Signbit and Two's complement
D	Decimal arithmetic

A decimal number is stored with one decimal digit per four bits.



The number is right justified and zero filled if necessary. Each digit (and sign) uses 4 bits, but the leftmost digit may occupy less than four bits if the word length is not a multiple of four. Such an incomplete field may cause a truncation of high order bits and should not be used to hold a digit other than zero. The sign is represented in the second rightmost 4 bit unit. The code for the digits is just their normal binary representation and the sign code is

1100 for + and 1101 for minus.

The remaining codes are also interpreted as a sign code with 1010, 1110 and 111 for + and 1011 for - .

4.1.1 Restrictions: $8 \leq n \leq 255$

REAL a bit string may be interpreted as a floating-point number (r) in many different ways, and the operation may be performed in a number of ways. It is not possible to include all possible interpretations in OBM, and beside base, exp and mant we have three different ways of representing the sign and three ways of truncation (trunc) plus actual.

The design of a real number is as follows:

bitstring:

s	e	a
---	---	---

number of bits: 1 exp-1 mant

Let $b = \text{base}$, $c = 2^{\text{exp}-2}$ and $\bar{a}$ the logical complement of a. The mantissa a is a binary number with the fictive binary point immediately in front of it.

The number is normalized, which means that $a \in [1/b, 1)$.

The sign has the following meaning:

sign = A

$$r = \begin{cases} \text{if } s=0 \text{ then } a \cdot b^{e-c} \\ \text{if } s=1 \text{ then } -\bar{a}b^{\bar{e}-c} \end{cases}$$

sign = B

$$r = \begin{cases} \text{if } s=0 \text{ then } a \cdot b^{e-c} \\ \text{if } s=1 \text{ then } -a \cdot b^{e-c} \end{cases}$$

sign=C

$$r = \begin{cases} \text{if } s=0 \text{ then } a \cdot b^{e-c} \\ \text{if } s=1 \text{ then } -\bar{a}b^{e-c} \end{cases}$$

Another important concept is the way of truncation, and this is given by trunc with the following meaning:

trunc	meaning
N	As actual computer
C	Truncating
R	Rounding
P	Parity arithmetic

4.1.1

The three ways of arithmetic (C,R,P) will now be further described. Let us look upon the number

$$U = \pm 0.U_1 U_2 \dots * b^{e_u} = f_u * b^{e_u}, \quad U \neq 0$$

U rounded to t digits is $fl(U, t) = \pm 0.U_1 \dots U_t * b^{e_u}$

Truncating arithmetic. Here we just drop the last digits $U_{t+1} \dots$

Here the error $\in (-\text{sign}(U)b^{e_u-t}, 0]$

Rounding arithmetic. Here we will sometimes change

the $U_1 \dots U_t$ to $U_1 \dots U_t + 1$ depending on $U_{t+1} \dots$

The error $\in [-\frac{1}{2} \text{sign}(u)b^{e_u-t}, \frac{1}{2} \text{sign}(u)b^{e_u-t}]$

Parity arithmetic. This is only defined for even bases, and it is a bit more complicated than the other two.

U is said to be t -stable if and only if the integer

$U_1 U_2 \dots U_t + b$ is odd.

Then $fl(u, t) = \pm 0.U'_1 U'_2 \dots U'_t * b^{e_u}$

$$\text{where } U'_1 U'_2 \dots U'_t = \begin{cases} U_1 U_2 \dots U_t & \text{if } U \text{ is } t\text{-stable or} \\ & U_{t+1} U_{t+2} \dots = 0 \\ U_1 U_2 \dots U_t + 1 & \text{otherwise} \end{cases}$$

The error will here be the same as for rounding arithmetic, but the probabilistic mean error will be zero and this is not true for rounding.

Restrictions: base = 2, 4, 8, 16 and 10.

$$3 \leq \text{exp} \leq 64$$

$$2 \leq \text{mant} \leq 259$$



4.1.1

STRING

A string is described by the following parameters.

ch is the number of bits/character and used is the number of significant bits for the character.

word is the number of characters stored together as a compact unit, used for memory allocation and integer compatibility. alpha is a name of the alphabet and defined by the ALPHA statement.

The alphabet is a mapping of the character set into a code.

The string type is used for string definitions, code conversion during string handling operations and as a code specification for I/O statements.

NODE

Node must not be operands to arithmetical instructions, but the individual partial words may be used as operands. They have no type like integer, real, signrepresentation, etc. They are just bit strings. But as the type of arithmetic is determined by the type of the operator all operands are just bit strings. It is impossible to define an arithmetical instruction of node type.

Example

```
FIX:TYP INTEGER,32,N,C;
```

```
INT:TYP INTEGER,36,N,0;
```

This is the normal IBM and UNIVAC fix point integers.

```
S:TYP NODE,12,3; WORD 6,6,5; WORD,E 10,5*8; WORD,E 5,23,12;
```

This means that partial words 1,2,3 will have a length of at least 6,6 and 5 bits. The partial words 4-9 will have the exact lengths 10,8.....8 and stored compacted together and the partial words 10,11,12 will be stored compactly with the lengths 5,23 and 12. Notice that partial words 9 and 10 do not have to be "close" to each other in memory, because they are defined in different word statements.

4.1.1 alpha is an identifier used as a label on an alphastat, and is used to specify a code alphabete. (See 4.1.2!)

node is present only if tpar contains NODE. The number n of wordstat must be the n given in the tpar and the wordstat must define exactly pw partial words. (See above!)

4.1.2 It is possible to express a mapping of a character set into a numeric code. This is done by the ALPHA statement. But also in this mapping it must be possible to represent the characters, and this is done according to the standard ISO 646. Here the available character set is represented in an array, and each character is represented by position in this array. In the alpha statement the desired code is connected to a position.

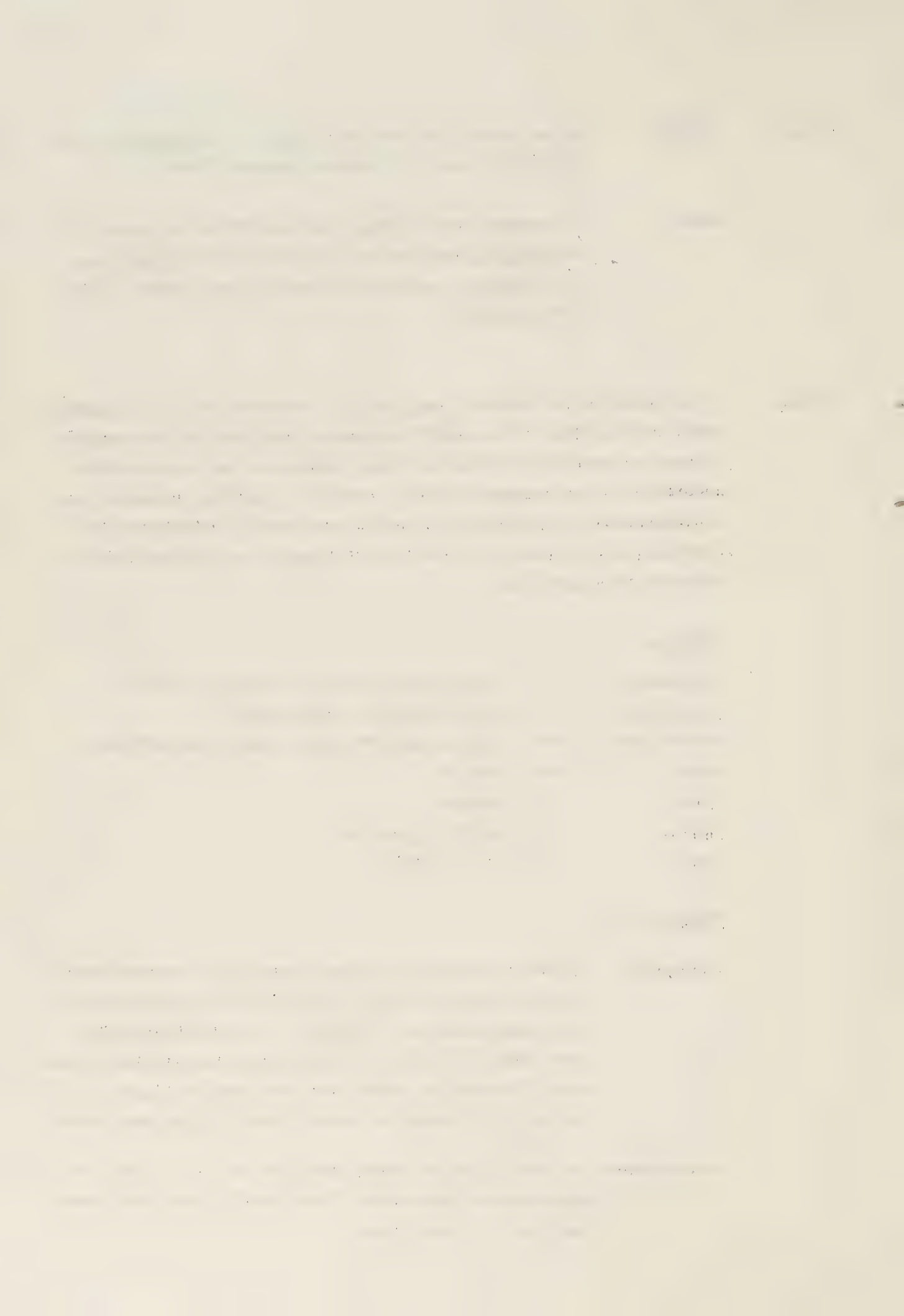
Syntax

<u>alphastat</u>	→	label:ALPHA,bits e^+ codelist del
<u>codelist</u>	→	codeelement{,codeelement} [*]
<u>codeelement</u>	→	code integer* code-code integer*code
<u>code</u>	→	integer
<u>bits</u>	→	integer
<u>radix</u>	→	RADIX e^+ rad del
<u>rad</u>	→	DEC OCT HEX

Semantics

alphastat is used to define the binary code of a character set. The used character set is the ISO 646, a character is here represented as a position in a character array (see special table). The positions in are ordered in the array from top to bottom and from left to right. The codelist gives the binary codes in the same order.

codeelement is used to define binary code for one or a number of consecutive characters ("positions"). The codeelement has four different forms:



4.1.2

- i/ code gives the code to one character.
- ii/ code-code gives a series of consecutive codes from code to code (limits included) to a number of characters.
- iii/ integer* gives the code 0(zero) to integer consecutive characters ("undefined").
- iv/ integer* code gives the same code to integer characters.

bits is the number of bits used to hold a character.

code is an integer by default in radix 10 notation, but code is sensitive to the radix pseudooperator.

radix is used to define input radix for integers. The radix may have the following values: DEC=10, OCT=8 and HEX=16. The default radix is 10 and the radix is in effect until the next radix statement. Only a very limited number of statements are sensitive for the radix setting.
{alphastat}

Example

The alpha statement will be explained by an example. As an example we will describe the definition of the UNIVAC Fielddata code.

```
FD:ALPHA,6 32*,5,45,1*,3,39,42,38,58,41,32,40,
           34,46,33,61,60,48-57,43,89,35-37,
           44,0,6-31,1,1*,2,34*;
```

The normal full character set (standard OBM) may be defined as

```
ISO:ALPHA,7 0-127;
```

The ISO 646 character array is international and position 66 should be printed as A and 92 as [.

Some countries have a slightly modified standard, to be able to print some characters in their national alphabet, this is hardware dependent (printer). In the Scandinavian and German standard the positions 66 and 92 will be printed as A and Ä.

ISO/R 646

Bits						Column							
b ₇ b ₆ b ₅ b ₄ b ₃ b ₂						Row							
0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	1	1	0	0	1	1	0	0	1	1	1
0	0	1	0	1	0	1	0	1	0	1	0	1	1
0	1	2	3	4	5	6	7						
0	0	0	0	0	0	NUL	TC ₁ (DLE)	SP	0	@	P		p
0	0	0	1	1	1	TC ₁ (SOH)	DC ₁	!	1	A	Q	a	q
0	0	1	0	2	2	TC ₁ (STX)	DC ₁	"	2	B	R	b	r
0	0	1	1	3	3	TC ₁ (ETX)	DC ₁	#	3	C	S	c	s
0	1	0	0	4	4	TC ₁ (EOT)	DC ₁	\$	4	D	T	d	t
0	1	0	1	5	5	TC ₁ (ENO)	TC ₁ (ENQ)	%	5	E	U	e	u
0	1	1	0	6	6	TC ₁ (ACK)	TC ₁ (ENM)	&	6	F	V	f	v
0	1	1	1	7	7	DEL	TC ₁ (ETB)	'	7	G	W	g	w
1	0	0	0	8	8	FE ₁ (FS)	CAN	(	8	H	X	h	x
1	0	0	1	9	9	FE ₁ (HS)	EM	)	9	I	Y	i	y
1	0	1	0	10	10	FE ₁ (LF)	SUB	*	:	J	Z	j	z
1	0	1	1	11	11	FE ₁ (VT)	ESC	+	;	K	[	k	{
1	1	0	0	12	12	FE ₁ (FF)	IS ₁ (FS)	,	<	L	\	l	
1	1	0	1	13	13	FE ₁ (ER)	IS ₁ (GS)	-	=	M	]	m	}
1	1	1	0	14	14	SO	IS ₁ (RS)	.	>	N	^	n	~
1	1	1	1	15	15	SI	IS ₁ (US)	/	?	O	_	o	DEL

CHARACTER CODES

CPU CODE (OCTAL)	80-COLUMN CARD CODE	HIGH SPEED PRINTER SYMBOL	DISPLAY CONSOLE TYPE 4009			OPERATOR'S CONTROL CONSOLE TYPE 3004	
			KEYBOARD SYMBOL	CRT SYMBOL	PAGE WRITER SYMBOL	KEYBOARD SYMBOL	PRINTER SYMBOL
00	7-8		//	Note 1	Note 1	K	//
01	12-5-8	{	{	{	{	UC	{
02	11-5-8	}	}	}	}	LC	}
03	12-7-8	a	ML (newline)	Note 2	Note 3	LF	LF
04	11-7-8	A	(no key)	Note 1	Note 1	RETURN	CR
05	(blank)	(space)	(space bar)	(blank)	(blank)	(space bar)	(space)
06	12-1	A					A
07	12-2	B					B
10	12-3	C					C
11	12-4	D					D
12	12-5	E					E
13	12-6	F					F
14	12-7	G					G
15	12-8	H					H
16	12-9	I					I
17	11-1	J					J
20	11-2	K					K
21	11-3	L					L
22	11-4	M					M
23	11-5	N					N
24	11-6	O					O
25	11-7	P					P
26	11-8	Q					Q
27	11-9	R					R
30	0-2	S					S
31	0-3	T					T
32	0-4	U					U
33	0-5	V					V
34	0-6	W					W
35	0-7	X					X
36	0-8	Y					Y
37	0-9	Z					Z
40	12-4-8	[					[
41	11						
42	12						
43	12-7-8						
44	1-8						
45	0-8						
46	2-8						
47	11-3-8						
50	11-4-8						
51	0-4-8						
52	0-5-8						
53	5-8						
54	12-0						
55	11-0						
56	0-3-8	(comma)					(comma)
57	0-6-8						
60	0	0	0	0	0	0	0
61	1	1					1
62	2	2					2
63	3	3					3
64	4	4					4
65	5	5					5
66	6	6					6
67	7	7					7
70	8	8					8
71	9	9					9
72	4-8	(apos.)					(apos.)
73	11-6-8						
74	0-1						
75	12-3-8	(period)					(period)
76	0-7-8					spec	
77	0-2-8	2 or stop					

Note 1: When either 00₈ or 04₈ code is received for either the CRT or the PAGEWRITER, it is completely ignored.

Note 2: When the 03₈ code is received for the CRT, the CRT addressing circuits are set to display the symbol for the next character code as the first symbol on the next line.

Note 3: When the 03₈ code is received for the PAGEWRITER, a combination carriage return and line feed action is initiated at the PAGEWRITER.

4.2 MEMORY ALLOCATION

As it is possible to have words of different length and structure in the same program and segment, we must in some way separate memory areas of different structure, else indexing will lead to chaos.

A memory area is defined to be of a certain length and structure.

An OBM memory area is a number of OBM words of a given length and structure. The allocation is dependent on storage property of used type. The only properties of an OBM word (cell) is a bit length and sometimes a partial word structure. Properties like REAL, INTEGER, etc, are not attached to the word, even if a REAL type is used to define the memory area.

Syntax

poolstat	→	label:POOL,type ϵ^+ {integer pool} $\{L, C R\}$ de
stringstat	→	label:STRING{,stopt} ϵ^+ chars, bits del
common	→	label:COMMON ϵ^+ pool{,pool}* del
pool	→	id
stopt	→	X R
chars	→	integer

Semantics

poolstat is a statement used to define the structure of a memory area and to allocate memory for it. label is defined to be the name of the area, this name cannot be used to reference data in the area. The structure of the area is given by type (only the storage properties of type is used). The size of the area will be integer words of the given type. If pool is given the area will be allocated the same memory as the area pool. The size will then be determined from pool. For the second method the type and the type of pool must be compatible (See page 26). If R is included in the instruction the area will not be physically allocated, because the memory area for this pool is allocated in another segment. The area in the other segment must have been defined by means of the same type and the same size (the name is not important).

4.2

If C option is included the area will be physically allocated in a common area that may be used for inter-activity communication (see 7.2.1). The common area and data words in this area is referenced in a normal way, but the area must be assigned to this activity.

Restrictions: number of words in pool $\leq 2^{15}-1 = 32767$.

the number of bits in pool $\leq 1\ 048\ 544$.

pool is an identifier which has appeared as a label on a poolstat.

stringstat is a statement used to define a string area. This is a memory area that has a restricted use. It may only be used to hold strings. It is a memory area suitable to hold a string, of chars characters. Each character has the given number of bits. The memory allocation for this area is machine dependent, but this dependence is invisible as long as the area is manipulated by means of string instructions. If R option is specified, then the area is allocated in another segment and if X option is specified, then the area is referencable from other segments. If option is omitted, the area is global and visible only within the segment.

Restrictions: $2 \leq \text{bits} \leq 8$

$1 \leq \text{chars} \leq 64000$

stopt is used to specify allocation.

chars is used to define the memory size.

common is used to define an area to be referencable from other activities. (See 7.2.1) The common area consists of a number of pools that must have been defined with the C option.

The pools included in a common statement may be defined in different segments, and in other segments than the segment containing the common statement. The pools in the common area may be referenced from different activities but in all activities referencing the common area, the common area must be defined with the same structure and name.

4.2 Restrictions on pool equivalence

When more than one pool is defined to use the same memory, but with different structures there must be some restrictions on the structures. The original pool say P0 must be defined in the "normal" way.

P0:POOL, T0 n;

T0 is a type with a bit length (final length after S(T0) has been considered) L(T0) and a store option S(T0) void, N, W, S, M, E and with a kind of type K(T0) INTEGER, REAL, STRING, NODE, BIT

Let us then define another pool P1 in the same memory area:

P1:POOL, T1 P0;

For the second statement to be valid there must be some restrictions on T1 depending on T0. Further $K(T0) \neq \text{NODE}$ and $K(T1) \neq \text{NODE}$ must be valid,

Properties of T0	Demanded properties of T1
$S(T0)=E$	$S(T1)=E$
$S(T0) \neq E$	$S(T1) \neq E$ $L(T0)=q L(T1)$ for some integer q.

It must further be noticed that the storage of P1 is not directly determined by T1, but rather by T0 and influenced by T1.

4.3 VARIABLES AND CONSTANTS

By means of the POOL-statement a memory area for the storage of data words is allocated. For convenient addressing it must be possible to name words in this area. When a data word is addressed by means of indexing, a check must be made to guarantee that the pool is not left in an uncontrolled way. This checking is timeconsuming and therefore should be performed under programmer control. There is a statement (checkop) to control this.

The allocated memory is initially zerofilled but it is possible to give other initial values for certain words.

Syntax

datastat.	→	label:DATA ϵ^+ pool{(X)},rel del label:DATA,type ϵ^+ pool{(X)},rel,value del
rel	→	integer * + integer * - integer
value	→	int float stringval struc
int	→	{+ -} ₀ ¹ nonzero digit*
float	→	int.digit ⁺ {+ -}nonzero digit*
stringval	→	'character' ⁺
struc	→	val {,val} [*]
val	→	int integer*int
checkop	→	INDCH,{ON OFF} ϵ^+ data{,data} [*] del
character	→	a position in ISO646 scheme "

Semantics

datastat The first type of the data statement will give a name to an OBM word with zero as start value. The word will be allocated in the area pool and with relative address rel. The relative addresses will start from 0 and must be less than the length of the pool.

If the *construction is used the * sign means the last allocated word in the pool in a preceeding data statement for that pool. It is possible to have any number of names for the same word. If the pool (X) construction is used, the data word shall be externally accessable.

4.3	datastat	The second type of the data statement will give a start value to the OBM word. The parameter <u>type</u> gives the type of <u>value</u>, <u>type</u> must be compatible (same word length) with the type of the pool. If <u>type</u> is a nodetype the value must be structured with the same number of partial words as <u>type</u>. If the <u>type</u> is an integer an <u>int</u> must be given as <u>value</u>, if it is a real <u>float</u> must be used and for string <u>stringval</u> must be used.
	int	is a normal integer and if the size of the given <u>value</u> makes it impossible to store according to <u>type</u> it is an error.
	float	is a decimal real number given with or without a scale factor (power of ten). The least significant digits are truncated in order to fit the given <u>type</u> and word length.
	stringval	is a string of characters. The string may consist of any character in the ISO646 alphabete.
	struc	is a structured value and each partial word in the node will be given an <u>int</u> as an initial value. If the construction <u>integer * int</u> is used the <u>int</u> will be repeated <u>integer</u> times. If an overflow occurs in at least one of the partial words it is an error.
	checkop	is a non-executable statement. It is a pseudo operation used to tell the translator to turn ON or OFF the generation of code for indexchecking during execution. This statement may be present anywhere within a segment. Index checking is turned ON or OFF for all the specified <u>data</u> words. This is valid for specified data word until it is changed by another checkop-statement for that data word.
	character	is a character from the ISO646 alphabete. The double string parenthesis " is used to express ' as a character. An empty stringvalue is impossible.

4.4 THE EQUIVALENCE STATEMENT

The equivalence statement is used to name a contiguous part of a structured word. The purpose is to provide means for a convenient addressing.

S y n t a x

equestat	→	label:EQU ϵ^+ stdata,pw1,pw2 del label:EQU,S ϵ^+ stdata,pw1 del
'pw1	→	integer
pw2	→	integer
stdata	→	id

S e m a n t i c s

equestat	The first form of the statement, defines <u>label</u> to be the name of a word consisting of the partial words pw1, pw1+1, --- pw2 in the structured word <u>stdata</u> . The included partial words must be a contiguous bit-string, that is defined in the same WORD,E statement. The second form is used to give a name to one single partial word in the structured word <u>stdata</u> .
pw1,pw2	must be integers that may be used as the index of an existing partial word in <u>stdata</u> .
stdata	is an identifier used as label on a DATA statement referencing a structured pool. If it is convenient to the programmer to have some sort of equivalence between densely packed words in a non-structured memory, the "pool equivalence" feature of the pool statement must be used and not the equivalence statement.

4.5 EXAMPLES OF DATA DEFINITIONS

Let us look upon some type definitions and some memory areas of this type, and how they may be allocated in some different computers.

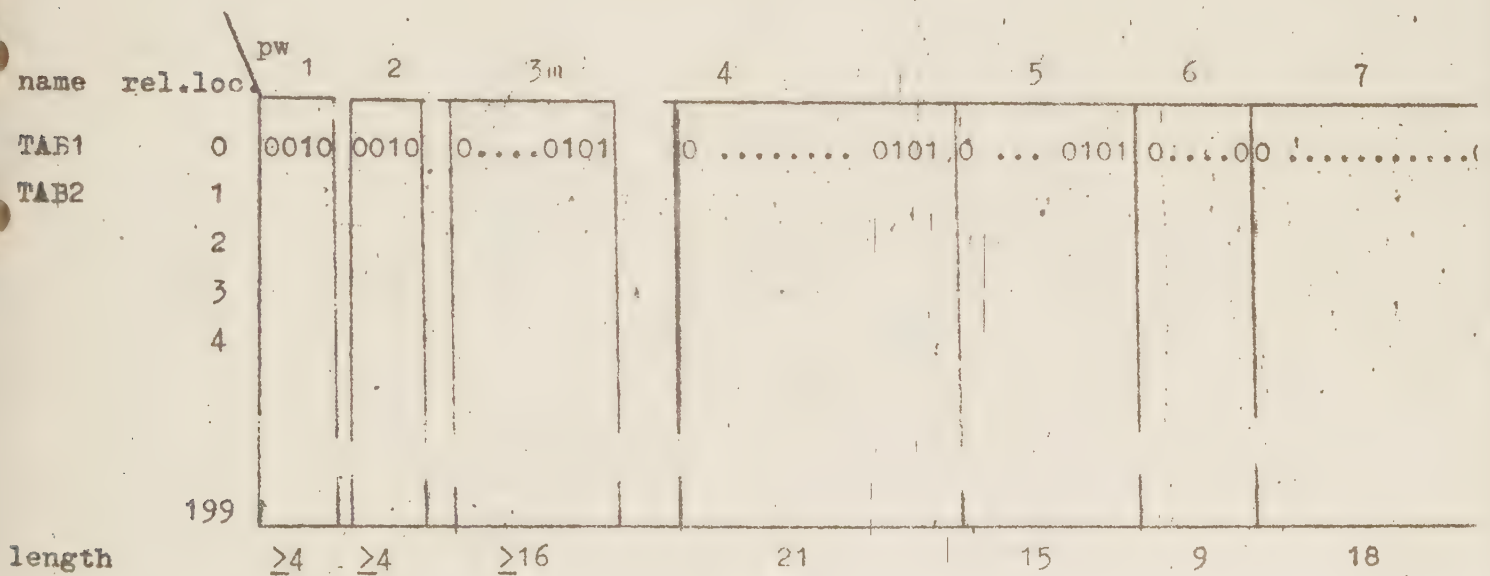
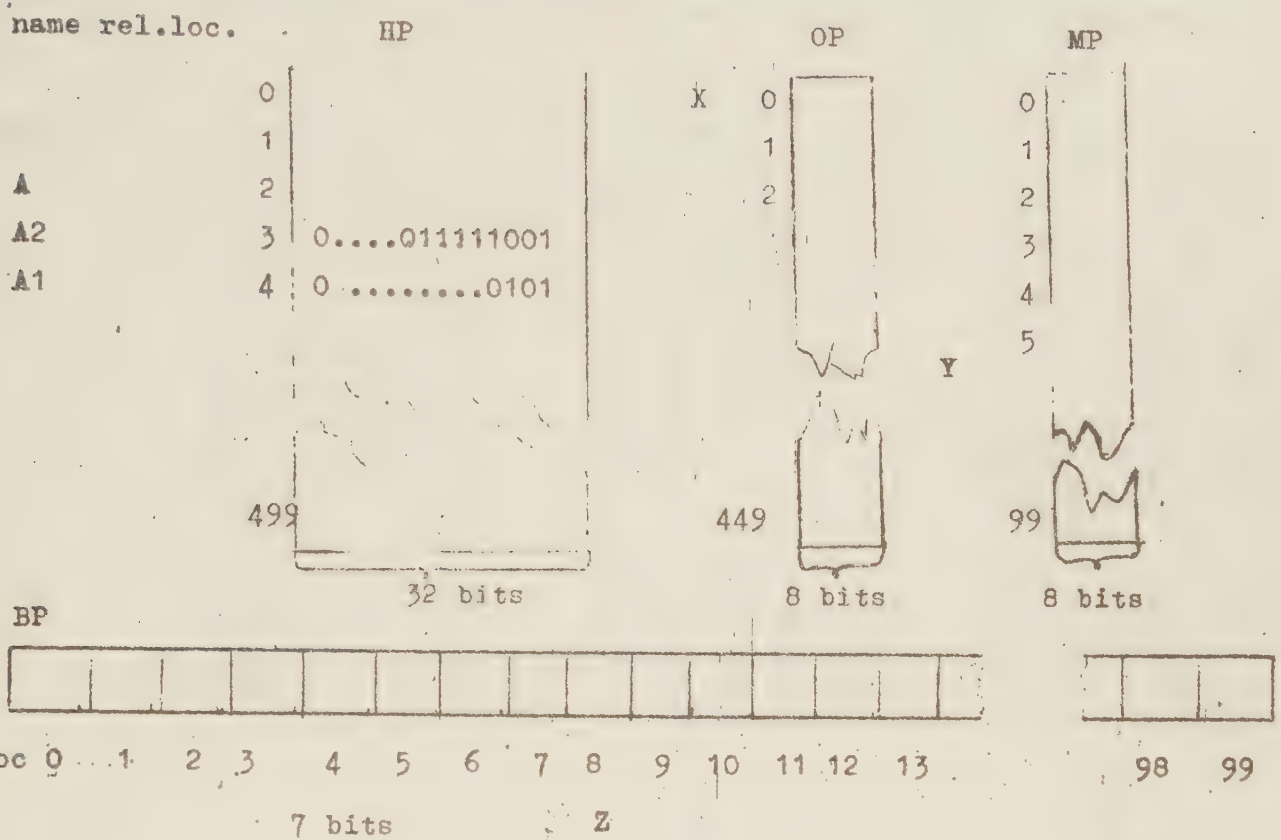
The OBM program part:

```
H:TYPE INTEGER,32,NO; HP:POOL,H 500;
Q:TYPE INTEGER,8,N; QP:POOL,Q 450;
B:TYPE INTEGER,7,E; BP:POOL,B 100;
M:TYPE INTEGER,8,W; MP:POOL,M 100;
A:DATA HP,2; A1:DATA,H HP, *+2,5;
X:DATA QP,0; Y:DATA MP,5; A2:DATA,Q HP,3,-6;
Z:DATA BP,8;
N:TYPE NODE,7,2; WORD 4,4,16; WORD,E 21,15,9,18;
TAB:POOL,N 200; TAB1:DATA,N TAB,0,2,2,3*5,0,0;
TAB2:DATA TAB,1;
```

This data definition part of an OBM program defines and allocates memory for five different memory areas. The storage of these pools are more or less implementation dependent. How these may be implemented on some computers will be shown.

Storage implementation will be exemplified by the UNIVAC-1108 computer with a 36 bits wordlength and partial word addressing facilities, the IBM computer with bytes of 8 bits and word of 4 bytes (32 bits), and a typical mini-computer with 16 bit words and some byte instructions at assembly level.

O B M First we may see how these memories will appear to the OBM programmer.



But notice that the OBM programmer does not know how this is really stored in the actual computer. What matters is that the program works as if it was stored in cells as above.

I B M. 360/370 series.

name rel.loc.

HP

0	0- - - - 0
1	0 - - - - 0
2	0 - - - - 0
3	0 011111001
4	0- - - - 0101
5	0- - - - 0

A

A2

A1

499



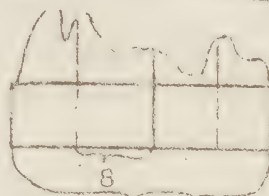
32

QT

X

0			
4			
8			
12			
16			
20			

448



32

MP

Y

0			
1			
2			
3			
4			
5			

99



32

BP

Z

32

TAB1

0	010	010	0- - 0101
	0- - 0101	0- -	
	101	0- - 0 0	- 0
1			

TAB2



- 4.5 An example of the "poolequivalence" is given. A construction that looks like the IBM-memory built up from bytes, and with a possibility to address bytes, halfwords, words and double words, is defined. A word consists of 4 bytes starting from a word boundary.

O B M - s t a t e m e n t s

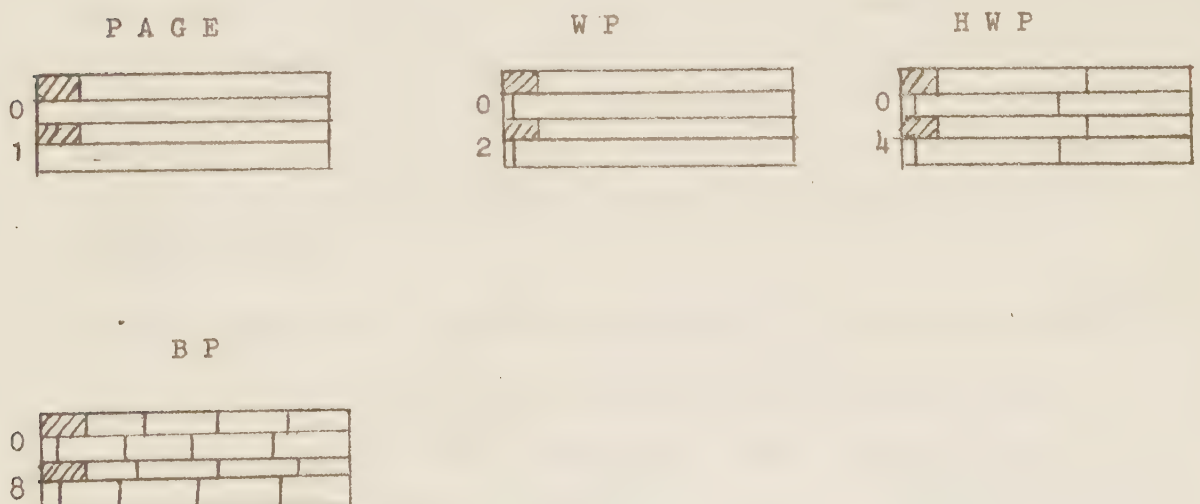
```

T1:TYPE INTEGER,64; T2:TYPE INTEGER,32; T3:TYPE INTEGER,16;
T4:TYPE INTEGER,8; PAGE:POOL,T1 512;
WP:POOL,T2 PAGE; HWP:POOL,T3 PAGE; BP:POOL,T4 PAGE;
PWORD:DATA PAGE,0; WORD:DATA WP,0;
HALF:DATA HWP,0; BYTE:DATA BP,0;

```

In this case all the pools will be allocated at the same location as PAGE. This means that the storage mode of the pools WP, HWP and BP is determined by the allocation of PAGE. If this is not possible or if the type of these pools describes another sort of storage this is an error. If you try to define the pools in term of the "smallest" BP instead of the "biggest" PAGE it will lead to an error, because the allocation may be impossible.

Let us look upon how this can be implemented on a UNIVAC 1108 computer. For a 32-bit and 16-bit word computer the implementation is trivial.



Notice that this storage may be inefficient as the equivalence pools are allocated storage as if they were a "partial" E store option.

5. INSTRUCTIONS

The OBM statements may be divided into different sets according to their logical use. One set consists of statements used to statically define memory and types. A small set consists of pseudo operation statements, used as directives to the translator. The third set is a set of statements that are called instructions. An instruction is a statement that may be executed. Some instructions are used to dynamically define memory, but most of the instructions are used to perform the "real work" in the program.

Important general concepts for instructions are addressing, scope of definitions and trimming of data.

5.1 GENERAL CONVENTIONS AND CONCEPTS

5.1.1 ADDRESSING

In O B M a word may be addressed by using its name with or without an index. The index may be absolute (a number) or variable, the same is true about the partial word index. In the O B M memory a great number of memory areas (pools) of almost any structure may exist. Indexing in such a complex memory must be very disciplined or else anything may happen (and it will). That means that all indices have to be checked, and if they will lead outside the memory pool it is an error. If the index is a variable the check must be done during run time, and this will be very time consuming, but it is possible to inhibit this check. A run time index error will give an index out of range interrupt (INDEX). The same is true for partial word indexing (PWIND).

The addressing will be explained by means of the following example.

Here A is a simple data word defined by a DATA or EQU statement, and B is a structured word, X and P are simple words. X and P must not contain more than 64 bits, and not more than 16 significant bits.

5.1.1	<u>Address</u>	<u>Meaning</u>
	A	A simple reference to word A.
	A(3)	This is references to the OBM-words A+3 and
	A(-2)	A-2. This is absolute indexing and checking may be done during translation.
	A(X)	This is a variable indexing and a reference to OBM-word A + contents of X. Here checking will be performed during run time.
	B(1,3)	Reference to partial word 3 in OBM word B+1. All indexing absolute.
	B(X,2)	Variable indexing and absolute partial word.
	B(3,P)	Absolute indexing and variable partial word.
	B(X,P)	The worst form of addressing, everyting is variable and checking may be very time consuming.
	B	This is a reference to a node consisting of a
	B(2)	number of partial words. This reference may
	B(X)	not be used for all instructions, only for some instructions. This will be further described for those instructions.

The following sort of "address" is an error: A(X(3))

In OBM it is possible to address many different kind of words, and words of a peculiar (not fit to computer) structure. This makes the addressing to a strong feature in OBM, but it may be inefficient. Therefore do not use more of the flexibility and strength than needed!

5.1.2 SCOPE OF DEFINITIONS

The normal scope of a definition is the segment. But within a segment loop and subroutine instructions may be used to surround OBM instructions and thus form a block like structure. In this report the word block will be used for such a structure, but it is not strictly a block in the algol sense. The loop instruction will be defined in 6.2 and the subroutine in 6.3.

The top level of a segment should not be regarded as a top block.

The scope of definitions may be global in the segment, external (referenceable from outside the segment) or local within the "block". Local labels may have the same name as other labels (on other levels) in the segment.

Datawords If not otherwise stated the datawords defined by the DATA statement are global. Such data words may be made external.

Data words defined as control indices or secondary indices in loops are local within the loop.

Instruction labels These are labels on executable instructions. Most labels on executable instructions may be jumped to, they are called transfer labels.

Transfer labels defined outside of all "blocks" are global and they may be made external. Transfer labels defined within a "block" are strictly local, and they may never be made external.

Instruction labels that are not transfer labels are always global.

Label variables It is also possible to define a label variable (by the LABEL statement), which is a variable intended to hold the value of a transfer label (external, global or local). If the label variable is defined outside of all blocks it is global but never external. If it is defined within a block it is strictly local. (To guarantee the block to be closed for jumps). If the label variable is local, it may only be given values that are local too.

5.1.2 Switch labels This is a label defined as the name of a switch, defined by the non-executable CASE statement.

If the switch is defined outside of all blocks it is global but not external, and the members of the label set used to define the switch may be global or external.

If the switch is defined within a block it is local and members of its label set must all be local.

How the references to data words and labels are satisfied is a very important property of the language. Therefore it must be strictly defined. If a reference is made outside of all blocks only the global and external variables will be searched to satisfy the reference. If the reference is made within a block, the search procedure is much more complicated.

Reference in a block

Data words It is possible to define local data words within a block. Those words are not contained in any pool. Local data words are always used as simple variables or as indices. Local data words within loops are called loop indices or secondary loop indices and such words may never be changed other than by the loop control. If they are used as a receiving variable it is an error and if a variable with the same name exists as global or external, this outer variable is used and changed.

Local data words in subroutines may be used as simple variables, and it is permitted to change their values, that means they may be used as receiving variables. It is always permitted to use global and external data words within a block.

Transfer labels These labels are always strictly local. They are
Label variables searched for within the block where the reference is
Switch labels made. No other blocks or top level are searched!

5.1.3 TRIMMING

The great difference between OBM and other assembly languages is the flexibility in specifying wordsize and in disposition of words. Most of the simple OBM instructions (like add) may be compared to a set of closely related instructions. We have a set of add instructions, containing one instruction for each possible type. In most programs we use only a small subset of this.

In higher languages you may have "mixed" arithmetic and in assembly languages all operands use to be interpreted to be of the same type and length. In OBM the type of arithmetic is completely defined by the instruction (and modifier), but the length of the operands may be different. The original type of the operands may also be different, but the instruction do not care about this type, only the type that defines the instruction. In order to use operands of different type (length) they must be trimmed (but not converted) before they are used. This trimming must be well defined, and here a definition is given.

An operand (A) is trimmed from one length (l_1) to another (l_2). The operand may have two different ways of sign representation, only signbit or signbit with complement (of some kind).

If $l_1 = l_2$ nothing will happen.

If $l_1 > l_2$ then A is cut down to l_2 bits and the leftmost part is discarded. Here we don't care about the signbit.

If $l_1 < l_2$ then A must be expanded (at left) to l_2 bits.

If we have sign with complement, we will add signbits at the left.

If we have only special signbit (for l_2), this will be moved left and zeroes will be inserted (to fill the gap). This trimming may only be meaningful for integers, but an operand of any type is trimmed exactly in the same way.

If A is a real number, this will give nonsense as result, but a perfectly well defined nonsense. For real numbers you should perform a conversation instruction in advance. To the trim function real numbers are like integers with signbit and complement.

5.2 INSTRUCTION PART OF SEGMENT

The OBM instructions repertoire may be divided into three main groups. These are

- i) Ordinary assembly instructions.
- ii) Higher assembly instructions.
- iii) Operating system interface instructions.

The first group contains normal simple instructions like those in other assembly languages. Here we have arithmetical, logical, branching and data transferring instructions.

The second group contains block like or macro like instructions. Most of them may be defined by a set of instructions of group i), but for some computers they may be more efficiently implemented than a set of simpler instructions.

The third group contains instructions to control privileged instructions, I/O instructions, parallel activity handling and other OS routine handling. This makes the operating system to be an integrated part of OBM.

S y n t a x

instruction	→ simpleinstruction subroutine loop
simpleinstruction	→ arithmetical logical branch stackstring keystat search system call pseudo empty
empty	→ {label:} EMPTY data

S e m a n t i c s

Instructions can be executable and non-executable. Non-executable instructions are used to define something in a dynamical (but not repeated) way. The definitions in the data description part are all statical and performed during translation. Pseudo instructions are a form of directives to the translator, and they are not present during the execution of a program. The executable instructions are the part of the program that does the main work.

- 5.2 subroutine This is a sort of block. It is possible to transfer control to and from this block only by means of subroutin calls and return statements. Data may be transfered to (and from) the block as parameters or through global or external data words. Subroutines may not be nested. Subroutines are defined in 6.3 !
- loop This is another sort of block, that may be nested. The main purpose of this block is to perform regular loops and to facilitate index checking. Loops are defined in 6.2!
- simple- These are the units used to build the inner part of instructions blocks and the executable part of the segment. They are all statements.
- empty An empty instruction, used to put a label on.

General description:

Executable instructions can be of two kinds:

- 1) Instructions that may be jumped to.
- 2) Instructions impossible to jump to.

Statements used in the instruction part have the following general structure:

label:operator ,specification parameters;

On instructions to be jumped to the label is defined as a transfer label that may be jumped to. On other statements the label is ignored or used to define a name on something. The operator is used to specify the main action, and the the specification is used to further define this action. The specification is often a type. The number and design of the parameters, if any, are dependent on operator.

5.3 ARITHMETICAL AND LOGICAL INSTRUCTIONS

5.3.1 ARITHMETIC

S y n t a x

arithmetical	→	transfer compute shift
transfer	→	move abs neg
move	→	label:= ϵ^+ adr,adr,type,type del
abs	→	label:+=,type ϵ^+ adr,adr del
neg	→	label:=-,type ϵ^+ adr,adr del
compute	→	label:atmop,type ϵ^+ adr,adr,adr del label:REMAINDER ϵ^+ adr del
atmop	→	+ - * /
shift	→	label:shiftop,type ϵ^+ adr,adr,opnd del
shiftop	→	/+ /- //+ //+
opnd	→	adr integer
arithmfault	→	AFCN,{ON OFF} del

S e m a n t i c s

move	If <u>type</u> is not specified the second operand is trimmed to the length of the first operand and then transferred to the first operand. If type is specified the second operand is first converted from the first type to the second type and the result is then trimmed and transferred to the first operand as above.
abs	The second operand is trimmed to the length of the first, then the absolute value is calculated and transferred to the first operand. The sign representation is determined from <u>type</u> . This action will guarantee the result to be positive even if the trimming will cut off significant parts of the number.
neg	This is the same as abs, but instead of calculating the absolute value the sign is changed.

5.3.1 compute

Here the second and third operand will be trimmed to have the length specified by type. Then the atmop modified by type will be performed, the result trimmed to fit the first operand and then transferred to the first operand. The operands have no type, they are just bitstrings of some specified length.

atmop

The type is completely defined by the type of the operator. For integer divide the remainder is saved, and may be inspected by the REMAINDER statement. Overflow and division by zero will cause an interrupt. The REMAINDER statement will transfer the remainder from the most presently Performed integer division to the given adr. If no such division has been performed, the remainder is regarded to be zero. The sign of the remainder is the same as the sign of the quotient, but minus zero is never produced. The produced remainder will be trimmed to the length of adr before the transfer. The saved remainder is not destroyed by the REMAINDER statement.

shift

Here operand two is first trimmed to the length of type, the result is then shifted opnd number of bits. The result is then trimmed to fit into first operand and transferred to this. The above shifts means (from left to right):

Right logical, Left logical, Right circular and Left circular.

For circular shifts, the shifted word is regarded to have the length implied by type, circular shifts may be timeconsuming if the word lengths do not fit the actual host computer.

arithmfault

is used to turn ON or OFF the producing of code for arithmetic fault checking. This checking is ON as default, and it may be turned ON or OFF any number of times.

5.3.2 LOGIC

These are normal bit by bit logical instructions.

S y n t a x

logical $\rightarrow$ label: logop,type ϵ^+ adr,adr,adr del
 logop $\rightarrow$ AND | OR | XOR | MSKPCK | MSKUPC

S e m a n t i c s

'logical Here the second and third operands must be trimmed to have a length to type. The logop is performed between these operands. The operands are trimmed to fit into the first operand. The result is then transferred to the first operand. The first three are the normal logical and, or, exclusive or.

In MSKPCK (Mask out and Pack), the third operand is used as a mask.

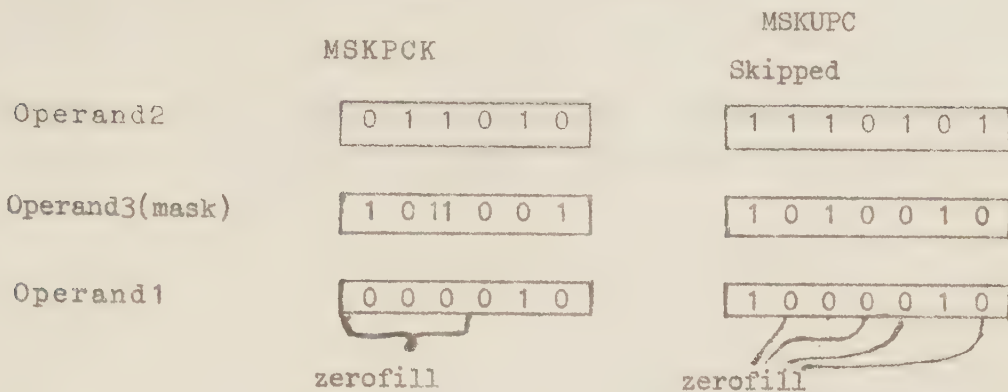
The corresponding bits of operand two is masked out and packed to the right, the left part is zerofilled. The result is then trimmed and transferred to the first operand.

The MSKUPC (Masked Unpack) operation is almost the inverse to this. The contents of operand two (right part) is distributed corresponding to the bits in the mask, the gaps are zerofilled and then the result is trimmed and transferred.

Ex a m p l e s:

	AND	OR	XOR	XOR
Operand 2	1 0 1 0	1 0 1 0	1 0 1 0	1 1 0 1 0
Operand 3	0 1 0	0 1 0	0 1 0	1 0 0
Operand 1	0 0 0 1 0	1 1 0 1 0	1 1 0 0 0	0 0 1 1 0

5.3.2 Operands are trimmed as signed integers. Above signrepresentation with complement is assumed.



5.4 BRANCHING

5.4.1 BRANCHING IN OBM

OBM contains unconditional branching with remembrance of return address if wanted, conditional branching and definitions of a case, that may be used as a form of switch.

S y n t a x

branch → case | casend | labass | goto | test

S e m a n t i c s

case	is used to define a switch. See 5.4.2 !
casend	is used to return from a package of instructions jumped to by a switch. See 5.4.3 !
labass	is used to set a value to a label variable. This may be used as a form of indirect branching. See 5.4.2 !
goto	is the unconditional branch statement. See 5.4.3 !
test	is the conditional branch statements. See 5.4.4 !

5.4.2 CASE AND LABEL ASSIGNMENT

S y n t a x

`case` $\rightarrow$ `label:CASE  $\epsilon^+$  trlab{,trlab}* del`
`labass` $\rightarrow$ `{label;}LABEL  $\epsilon^+$  labvar,trlab del`
`labvar` $\rightarrow$ `id`

S e m a n t i c s

`case` this is a non-executable instruction, which defines a switch with the name given by label. A switch is an ordered non-empty set of labels that may be jumped to. The labels are numbered from one and upwards. The case statement may be present anywhere in the instruction part of a segment, but it must appear in the input stream to the translator before it is used as a switch. The labels within the switch may be external, but the switch itself is always local, or global.

`labass` This statement defines labvar as a variable able to hold a pointer to an executable statement that may be jumped to. The labvar is given the value trlab. The labvar may be redefined dynamically by any number of LABEL statements. The labvar may be used in a GO statement (but not in a CASE). It may be regarded (and implemented) as an indirect jump.

`labvar` The label (trlab) to give labvar its value may be external, but the labvar is never external - it is global or local.

5.4.3 UNCONDITIONAL BRANCH STATEMENTS

S y n t a x:

`goto` $\rightarrow$ `{label;}GO  $\epsilon^+$  {swlab(ind)|trlab|labvar}{,retlab} del`
`casend` $\rightarrow$ `{label;}CASEND del`
`retlab` $\rightarrow$ `trlab`
`swlab` $\rightarrow$ `id`

5.4.3 S e m a n t i c s

trlab	Is a transfer label.
labvar	Is a label variable.
swlab	Is a switch label, defined as a label on a case statement.
retlab	Is a transfer label.
goto	The goto statement performs an unconditional jump. If the switch construction is used then the jump address is the corresponding address in the label set. The labels in the set are numbered from 1 and upwards. If the value of ind is less than 1 or greater than the number of labels in the set it is an error. If ind is a number the error is flagged during translation, else if it is a variable index an error interrupt will occur. (Type:CASE). If the goto statement is performed within a block the switch and all its labels must be defined in the same block. The second construction, with transfer label, will perform a jump to given address. If the goto statement is within a block the label must be defined in the same block. The third construction, with the label variable, will perform a jump to the address stored in the label variable. If this is done within a block both the label variable and the value of it must be defined in the same block as the goto statement. If the retlab is given, the goto statement will register this as the return point for casend. If the goto statement is within a block the returnpoint registered is local for that block.
casend	The <u>casend</u> will cause a return to the registered returnpoint, if one is registered. If no returnpoint is registered the statement will have no effect. The returnpoint is always deregistered by casend. At most one returnpoint for the level may exist, no queueing. If the casend is performed within a block, the local return point will be used. At most one local return point in one block may exist, and the only way of registering it is by a goto within the block.

5.4.3 casend The casend statement may be used at the top level even if there is no goto with return point. But a casend within a block may only be used if there is some goto statement with return point within the block. It is not necessary to execute the goto statement. If the goto statement is not present there is no return point defined and the caseend statement is illegal. If the goto statement is present but not executed before the execution of the caseend a return point is defined in the block but no special value is given to it and the caseend statement is an empty statement.

5.4.4 TEST OR CONDITIONAL BRANCH STATEMENTS

OBM contains a number of instructions which may be related to the algol statement goto if cond then L1 else L2;

Syntax

```

test      → zerotest|comptest|membtest
zerotest  → {label:}ztop,type  $\epsilon^+$  adr,trlab{,trlab}del
ztop      → IF=0|IF<0|IF>0|IF><0
comptest  → {label:}top,type  $\epsilon^+$  adr,adr,trlab{,trlab}del
top       → IF=|IF<|IF>|IF><
membtest  → {label:}IFMEMB,type  $\epsilon^+$  adr,adr,adr,trlab
           {,trlab}del;
```

Semantics

type This defines the wordlength, maintype (integer,real,--), signrepresentation and word disposition for the instruction. All operands will be considered to be of this type, if not, the operands are trimmed in the same way as for arithmetic instructions.

adr This is the address of a data word of any type, but its length will be trimmed to the length defined by the type parameter.

trlab This is a transfer label (local, global or external). A switch (case) label is not permitted in a conditional branch statement.

trlab If the second trlab is omitted it is equivalent to "next instruction".

5.4.4 For all instructions, if the condition is satisfied the control is transferred to the statement at the first label else to the statement at the second label. The condition depends on both operator and type.

IF=0	The operand is compared to zero, the comparison is type dependent in the following way:
INTEGER	The condition is satisfied both for +0 and -0. This comparison is dependent on sign and number representation.
REAL	The condition is satisfied if the operand is exactly equal to zero according to the zero-representation of the specified type.
STRUCTURED WORD	Illegal type.
BIT	The condition is satisfied if all bits are zeroes.
STRING	Equivalent to BIT.
IF><0	The comparison is made in a way analogous to IF=0, but the condition is reversed.
IF<0	INTEGER The condition is true if the operand is negative and non-zero, according to sign representation.
	REAL The condition is true if the operand is negative and non-zero, according to the number and signrepresentation given by type.
	STRING Illegal type.
	STRUCTURED WORD Illegal type.
	BIT Illegal type.
IF>0	Same as for IF<0, but negative must be changed to positive.
IF=	This instruction is valid for INTEGER, REAL, STRING and BIT, but STRUCTURED WORD is an illegal type. The condition is true if the two operands are exactly equal bit by bit. This means that +0 and -0 are not equal, if it is possible to have different bit string representation for these.

5.4.4 IF>< Analogous to IF= .

IF< The two operands are compared according to type.

INTEGER A normal comparison, but notice that +0>-0 !

REAL A normal comparison according to type.

STRING The contents of the operands are regarded as unsigned integers. If the codes of the characters are in a reasonable order in the alphabete, this means that the strings may be sorted in an alphabetic order.

STRUCTURED
WORD Illegal type.

BIT Operands considered as unsigned integers (binary).

IF> As IF< .

IFMEMB The condition is satisfied if the first operand is contained in the closed interval defined by the other two operands. It is typedependent in the same way as for IF< and IF>.

6. HIGHER LEVELLED INSTRUCTIONS

6.1 STRING AND STACK HANDLING

OEM contains special facilities for defining stacks and strings and for manipulation of these.

6.1.1 STRINGHANDLING

S y n t a x

stackstring	→	strdef strhdl stkdef stkhdl
strdef	→	label:DEFINE,STRING ϵ^+ string,size,type,finish del
string	→	id
size	→	integer adr
finish	→	true
strhdl	→	get chmove getptr setptr pack unpack
get	→	{label:}GETCH ϵ^+ str,adr del
chmove	→	{label:}CMOVE{,CONV} ϵ^+ str(ind),str(ind),num del
str	→	id
num	→	integer adr
getptr	→	{label:}GETPTR ϵ^+ {str stk},adr del
setptr	→	{label:}SETPTR ϵ^+ {str stk},adr del
pack	→	{label:}PACK ϵ^+ str(ind),num,adr del
unpack	→	{label:}UNPACK ϵ^+ str(ind)num,adr del

S e m a n t i c s

strdef In the data definition part of OEM it is possible to define a stringarea, and to reserve a memory area for this. The stringarea can be global or external. The STRING statement just sets up the area with a certain structure (machine dependent) suitable to hold string data. In order to manipulate this area as a string - a string must be defined in an executable statement. The strdef statement defines a string with a name given by label, and allocated in the string area given by area. This area must have been defined by a STRING statement. The size of the string is given in characters by size, this size must not be greater than the size of the area.

6.1.1 strdef

The characters are all of a type given by type, and this type must have a character length not greater than the length used to define the area. The type must of course be of string type with an alphabete. The last parameter finish is a transfer label used by other stringhandling instructions.

The DEFINE statement is an executable instruction. When it is executed a packet will be set up with information needed for stringhandling. The string will also be space-filled and a character pointer will be set to point at the first character (pointer = 1). Notice that the code used for spaces is determined by type. The same DEFINE statement may be executed more than once and the size may be changed from one execution to another.

get

This instruction is used to get the next character from a string. The character is transferred to adr - this referenced word must be big enough to hold the character, that will be right justified in the word. The rest of the word will be zerofilled.

The string must have been defined in advance and the string pointer will be incremented one step. If the string end is reached no transfer and pointer update will occur, but the control will be transferred to the defined finish label for the string.

chmove

This instruction will move one portion of a string into another string. The start of the substring to be moved is given by ind and the destination by the ind for the second string. The number of characters to be moved is given by num. If the number of characters to move will cause a reference outside of the two strings, then an out-of-string interrupt will occur (STACK See 9!)

If CONV option is omitted the move is just a block transfer and the two areas for the strings must be of the same structure (number of bits per character). If CONV is specified a conversion from the type associated to the first string to the type of the second string will take place. If conversion is used the two string areas may of course be of different structure.

- 6.1.1 **getptr** The actual value of the string pointer (for str) will be transferred to the word referenced by adr. The transferred value will be trimmed to the length of the addressed word. This instruction is also valid for a stack.
- setptr** This is the reverse of getptr. The contents of the addressed word (adr) is considered to be an integer without sign (bit string). If this instruction tries to give a value outside string dimension to the pointer, an out-of-string interrupt will occur (STACK), and the pointer is not changed. This instruction is also valid for a stack.
- pack** A portion of the string starting at ind and containing num characters will be packed into data words and stored in the buffer starting at adr. The length of the data words must be equal to $ch * words$ from the type used to define the string. A pool overflow will lead to an index-error interrupt (INDEX).
- PACK may be used to achieve a sort of compatibility between strings and integers or bit strings. If the string will not fill the last used word in the buffer completely, the buffer is left justified and zerofilled.
- unpack** This is the reverse of PACK. If the specified string portion is too small an out-of-string interrupt will occur (STACK).

Strings and input/output

It is possible to transfer characters from a text file to a string by the READ instruction. To transfer characters from a string to a text file the WRITE instruction is used. It must be noticed that the number of bits really used (and transferred by the I/O-instruction) to hold a character is machine dependent. Therefore care must be taken when using text files.

In order to transfer strings to and from files in a completely machine independent way, binary files should be used. The PACK (or UNPACK) instruction may be used to convert a string into (from) binary words in a buffer. The I/O instructions are further described in chapter 9.

6.1.2 STACK HANDLING

S y n t a x

stkdef	→	label:DEFINE,STACK ϵ^+ stkbasesize,underflow,overflow del
stkbasesize	→	data data(ind)
underflow	→	trlab
overflow	→	trlab
stkhdl	→	stack pop top getptr setptr
stack	→	{label:}STACK ϵ^+ stk,adr del
pop	→	{label:}POP ϵ^+ stk,adr del
top	→	{label:}TOP ϵ^+ stk,adr del
stk	→	id

S e m a n t i c s

stkdef is used to define label as the name of a stack. The bottom of the stack is at the data address stkbasesize, and the stack size is given by size. The pool where the base is located must be big enough to hold the stack.

underflow and overflow are transfer labels, where to jump, if under- or overflow occurs (during stack operations).

DEFINE creates a package to control the stack, a top pointer initially pointing under the bottom, (pointer=0). When the top pointer is zero it points at a non-existing element of the stack. This zeroth non-existing stack element is not included in the stack size and it may be outside the pool without being an error. The top pointer points at the top element of the stack. The stacks must be stored in the normally defined data areas of the program, and they are not protected against other useage.

stack is used to transfer the contents of the second operand to the top of the stack. Test for overflow and update pointer.

pop is used to transfer the contents of the top of the stack to adr. Test for underflow and update pointer.

top is used to transfer the contents of the top of the stack to adr. If an index is specified the element transferred is located ind elements under the top (top-index).

6.1.2 `getptr setptr` These instructions have the same meaning as for strings.

`stk` is used to name a stack. The id must have been defined on a stkdef statement.

6.2 LOOPS

OBM contains statements to define blocks suitable to repeated execution. Such loops may be nested to the depth of . Local data words may be defined within a loop-block, as index and secondary indices.

S y n t a x

```

loop  → {label:}REPEAT c+ index,start,step,fin del
        {SECIND c+ secind(index,numb){,secind(index,numb)}* del
        {exit|instruction}+ REPEND c+{var} del
        {label:}REPEAT,L c+ times del{exit|instruction}+ REPEND del

exit  → {label:}LEAVE c+{var} del
index → id
start → integer|adr
numb  → {+|-}_01 integer
step  → integer|adr
times → integer|adr
fin   → integer|adr
secind → id

```

S e m a n t i c s

repeat The simplest loop is the REPEAT,L loop. This is a block that will be executed the specified number of times. The other form (without L) will define a block that will be repeatedly executed, controlled by an index. This is a "step until loop". A change of the values of start, step and fin during the loop is permitted, but will have no effect upon the loop control. The index may not be changed during the loop, except by step in the loop control. The only way to leave the block is to repeat it until fin is reached or by the exit statement. In both cases the instruction following the loop-block will be executed. See also "scope of definition and block structure" in 5.1.2 !

6.2	<code>exit</code>	The only way to leave a loop-block in advance. This statement makes it possible to save the loop index value in <u>var</u> , when the block is left.
	<code>index</code>	is used as a variable local to the loop. It may be compared to high level languages "control variable" in loops. It is often used as an index, but may also be used as a data word, but not as a receiving ("left") variable.
	<code>start</code>	is used as a start value of index.
	<code>step</code>	is used as an increment (decrement) of index during looping.
	<code>fin</code>	is used as the final value of index.
	<code>secind</code>	is used as an extra local data word. Its value at the start of each repetition of the loop is determined from the value of index plus an integer offset.

6.3 SUBROUTINES

Most assembly languages do not include the subroutine feature, but they have instructions for branching and remember return address. These instructions may be used to implement subroutines, but the "subroutine" is just a part of the code that is not closed for other usage. High level languages have subroutines which cannot be entered in any other way than by a subroutine call. This later form of closed subroutines are included in OBM.

The main reason to include them is to build subroutine libraries. A subroutine in a library is written before and independent of the programs using the routine. The subroutine should be independent of the type and storage of its actual parameters as far as possible. Within a subroutine the formal parameters are regarded to have a certain type, but to the outside they have only a length in bits. The actual parameters are trimmed to have this length. A subroutine operates on its actual parameters much in the same way as an arithmetical OBM instruction operates on its operands.

- 6.3 If the subroutine is defined in another segment (most common) than where it is called, the subroutine must be declared external by the exsub statement in the calling segment (See 3.2!) A subroutine is always referencable from other segments.

S y n t a x

subroutine	→	nonrec rec foreign
nonrec	→	label:NONREC ϵ^+ paramlist del {simpleinstruction loop return} ⁺ ROUTEND del
rec	→	label:REC ϵ^+ paramlist del {LOCAL ϵ^+ loclist del} [*] {simpleinstruction loop return} ⁺ ROUTEND del
recursion	→	RECLIM ϵ^+ integer del
foreign	→	label:FSUB ϵ^+ fsubdescr del
paramlist	→	param{,param} [*]
param	→	data (lng,ptype)
lng	→	integer type
ptype	→	0 1 2
return	→	{label:}RETURN del
fsubdescr	→	(wn,wl) {,(wn,wl)} [*]
call	→	{label:}CALL,sub ϵ^+ actparlist del
actparlist	→	{integer adr} {,{integer adr}} [*]
sub	→	id
loclist	→	id {,id} [*]
wn	→	integer
wl	→	integer

6.3 S e m a n t i c s

- nonrec This is a non-recursive subroutine. It is possible to call other subroutines from a non-recursive subroutine, but this is not permitted if a chain of calls may lead to an indirect (or direct) recursion.

- rec This is the definition of a recursive subroutine. The call of a recursive subroutine will push a block of data into a special subroutine stack. This block contains the return address, all local parameters and the contents or addresses of the actual parameters. The subroutine stack has a finite length and the depth of recursion is limited. Restriction on subroutine stack length may also be set by pseudo instruction (RECLIM). A return from a recursive subroutine will also pop the stack.

- loclist The LOCAL statement defines a set of local data words that may be used as internal storage in a recursive subroutine.

- recursion This statement is used to define the size of the subroutine stack, other than the default size. The size is given in bytes, on a non-byte oriented machine the stack size may be greater than stated. It is difficult to translate a byte size to a depth of recursion. This translation is dependent on the number and size of the subroutine parameters and local variables. It is also implementation dependent. The default value is 1024 bytes and the value given in a recursion statement must satisfy

$$S \geq \sum_{i=1}^n (P_i + L_i) + 1$$

A great stack size may force the computer to use a virtual stack, if the virtual part of this stack is actually used, which is very time consuming.

6.3 foreign

This is a possibility to link to a machine dependent piece of code written in any language. It is the responsibility of the programmer that this foreign subroutine will perform the desired action, and that the return will be correct.

It is not possible to define the semantics of a foreign routine completely within OBM. This is implementation dependent, and foreign subroutines should not be used without a good knowledge of the actual OBM implementation. This is not intended for general use.

fsubdescr

This is used to describe the interface between OBM actual parameters and the foreign routine. An information package containing return address and parameters is passed to the subroutine, and back to the OBM program. This is implementation dependent. Each parameter is described by the pair (wn,wl). wl is the wordlength of the machine, or for byte oriented machines it may be a multiple of eight. wn is the number of such machine words used to hold one parameter.

param

The actual parameter in a call does not have to be defined as external. This means that the same subroutine may be used for different types of its parameters. During the call the bit length must be passed to the routine to be checked. When the routine is defined the lengths of its parameters is given by lng. When the routine is called the actual parameters will be trimmed to this length. lng may be given by an implementation dependent type. (See 4.1)!

For trimming the signrepresentation is determined from the type. If word length is given as an integer (and not type), the default signrepresentation is with complement.

6.3

ptype	addressing mode
0	value
1	reference
2	absolute number

Addressing by value means that only the value of the parameter is important. The value will be fetched at the start of the routine and cannot be changed during execution of the subroutine. Addressing by reference is more expensive and should only be used for result parameters. That the parameter is an absolute number means that the actual parameter is a positive number given in the call of the routine. It may be regarded as a bit string.

The subroutines are always external, but if they are not defined in the calling segment they must be declared by the SUBREF statement in the calling segment.

- sub The label on a rec, nonrec or fsub statement is defined as the name of the entry to a subroutine.
- call A subroutine is executed by means of a CALL statement.
- actparlist The actual parameters are connected to the corresponding formal parameters. The actual parameters may be of any type, but they will be trimmed to the defined length of the formal parameters.

6.4 SEARCH INSTRUCTIONS

In OBM a number of search instructions are included. It is possible to search a data area of a given length, according to a key.

S y n t a x

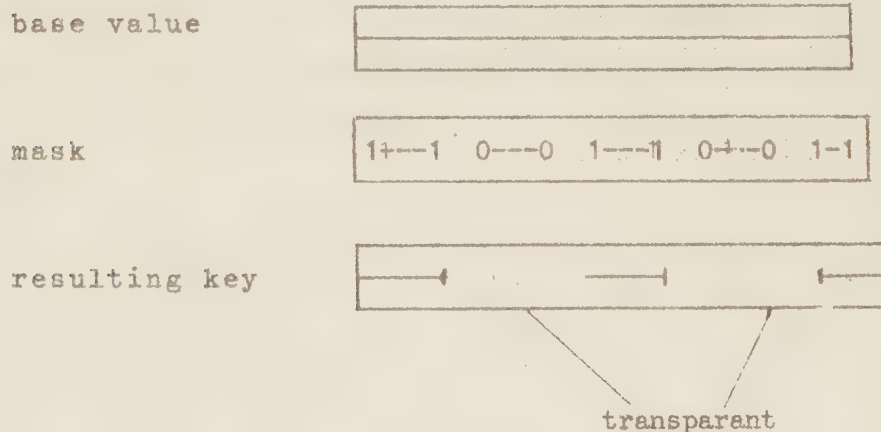
```

keystat  → label:KEY,type  $\epsilon^+$  data{(ind)} ,mask del
mask     → adr integer {,integer}*
search   → {label:}srchop  $\epsilon^+$  adr,data{(ind)},key,number,nofind del
srchop   → SRCH=|SRCH>|SRCH<|SRCH<<|SRCHZ
key       → id
number   → adr|integer
nofind    → id

```

S e m a n t i c s

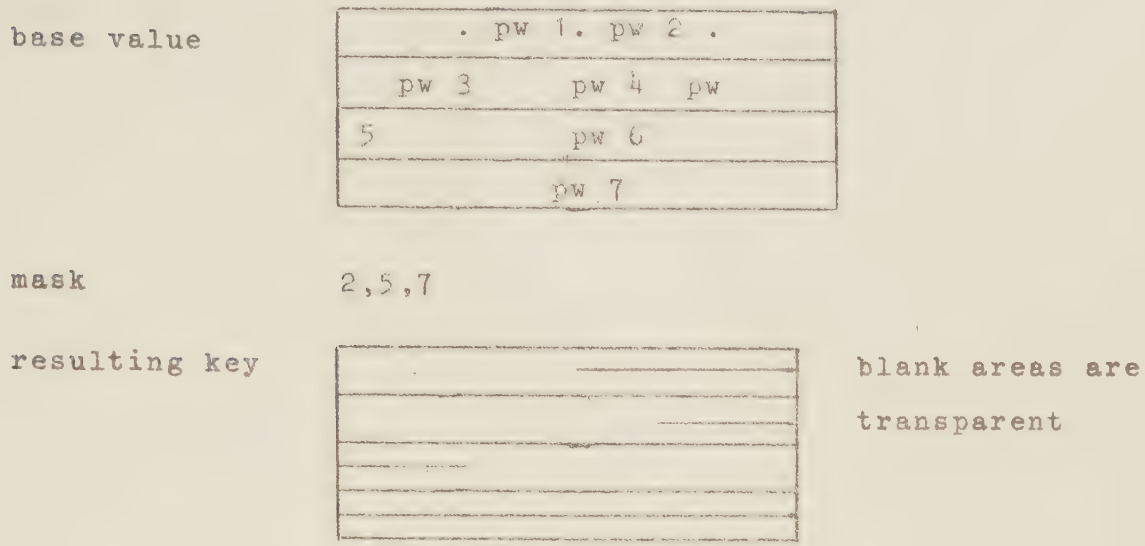
keystat is used to define a key used during a search operation. The key is built from a base value and a mask. For non structured keys the mask is given by adr or integer and the key is built according to the following figure:



For structured keys the mask is a number of integers in ascending order. This set of integers is used to specify which indices shall be included in the key. For structured keys the base value must be structured, with the same structure as key.

6.4 The structure and length of key is determined from type on the key statement.

The following figure gives an example of a structured key:



The key stat is executable and until it is executed the transparent parts of key contains zero.

mask is used to tell what parts of the base value that should be used as the key. For structured keys the mask may contain a set of integers, and the greatest must not exceed the number of partial words in the type. For non structured keys the mask must be given by adr or one integer.

search is used to search a data area according to a specified condition. The part of a pool starting at data (ind) and with the size number words is regarded as the area to search. This search area must be completely within the pool. If the condition is satisfied the relative word address within the search area is returned in adr. If the condition is not satisfied within the search area a branch to nofind will be made. The comparison is made between the words in the search area and the key, in that order. During the comparison the transparent nodes of the key and the corresponding parts of the search area words are completely ignored. The transparent parts are regarded as "non-existing" and the rest is considered to be a unsigned integer. For SRCHZ the contents of the non-transparent parts of the key are ignored.

6.4 key is an identifier that has appeared as a label on a key-stat.

7. OPERATING SYSTEM COMMUNICATION INSTRUCTIONS

Most computers have a set of instructions that are executable only in a privileged mode. Among these are the I/O instructions.

If these instructions should be of any use the unprivileged user must have access to the service of these instructions. This access is made through the operating system. The user may call an operating system service routine which is run in privileged mode.

OBM contains instructions that performs such service that normally will be performed by operating system routines. These instructions will be implemented by help of the OBMOS operating system. This part of the operating system is an integrated part of OBM.

7.1 SYSTEM INSTRUCTIONS

S y n t a x

```

system    →  exprim|inout|devctrl
exprim    →  exec|fork|nact|aexit|err|abort|wait|passive|
              active|ifact|ifterm|iferror|alock|aunlock|chain|
              link|up

```

S e m a n t i c s

```

exprim    These instructions are used to control the execution of
           programs and activities. Instructions for parallell execu-
           tion are included.

inout     These instructions are used to control the data transfer
           between the primary memory and the secondary memories or
           I/O devices. (See 7.3).

devctrl   These instructions are used to control other functions of
           I/O devices than data transfers. (See 7.3!)

```


7.2 PROGRAMS AND ACTIVITIES

An absolute program is a piece of code and data in a format that makes it suitable for loading into the primary memory. A program has a name recognized by the OBMOS system.

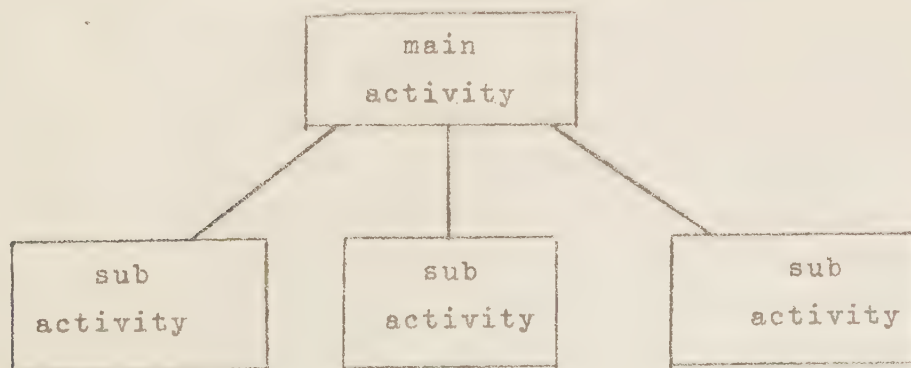
An activity is based on a loaded (may be swapped) program and an execution path or sequence control within this program is defined. More than one activity may load the same program, this can be done in two different ways. One method is to load the program twice and let each activity use an own copy of the program. The other method is to load it once and let the two activities be different execution paths in the same shared code (and data). The difference is that if the program is loaded twice both activities will have its own code and data, else the code and data will be shared and synchronization must be handled by the programmer.

A program may be executed by the operating system as a main activity. This may be done by an OBMOS command or from a user activity.

A main activity may start other parallell subactivities. Activities are normal and they may be synchronized and referenced from other activities belonging to the same main activity.

It is said above that an activity is based on one program. This is not the whole truth. An activity may be based on a chain of programs with an execution part. This means that programs may be used as sub-programs to other programs (or itself).

A main activity together with its subactivities is a parallell system. Within a parallell system the activities can be synchronized. A job may contain a number of such parallell systems, but there is no synchronization between such systems.



A parallell system.

7.2 The main activity in parallel system may be terminated before the termination of any subactivity, and the system will still be alive (without a living main activity).

7.2.1 EXECUTION PRIMITIVES

S y n t a x

```

exec      → {label:}EXEC  $\epsilon^+$  absname del
fork      → {label:}FORK  $\epsilon^+$  actname,absname{,dataarea} del
nact      → {label:}NEWACT  $\epsilon^+$  actname,trlab del
aexit     → {label:}EXIT del
err       → {label:}ERROR del
abort     → {label:}ABORT del
absname   → id
actname   → id
dataarea  → id

```

S e m a n t i c s

exec is used to start a program as a main activity. absname is the name of a program known to the system. This may also be done by an OBMOS command in the job stream. A main activity will always have the activity name MAIN.

fork is used to start a parallel subactivity. A subactivity is always belonging to a main activity- If fork is executed from a main activity the new activity belongs to this. If fork is executed from a subactivity the new activity belongs to the same main activity as the starting activity. This means that there are no "sub-subactivities". The new activity is given the name actname that must be unique within the same main activity system. The activity is based on a copy of the program absname, no code or data is shared except the special common dataarea. The sequence control starts always at the start address of the program.

- 7.2.1 nact** is used to start a parallel subactivity in (almost) the same way as fork. The difference is that no new copy of a program is used, but the same code and data as for the calling activity is used. The sequence control starts at trlab. A subactivity created by nact statement may be regarded as a new execution path within the same code. All protection of data areas must be done by the programmer himself.
- aexit** is a statement used to terminate the requesting activity or subactivity (and no other). The termination is registered by the operating system as "normal".
- err** is a statement used to terminate the requesting activity as for aexit above. But the termination is registered as an "error" by the operating system.
- abort** is a statement used to terminate all activities belonging to the same main activity as the requesting activity. Also the main activity is terminated (error termination).
- dataarea** is an area defined by a common statement. (See 4.21)
The common statement must be present in the segment using the fork statement. This area is intended to be used as a shared data area for different activities. The area must have been defined in the same way in all programs using the area. This area is assigned to only one activity at the time. From the beginning it is assigned to the main activity. This assignment is changed by the alock and aunlock statements.

7.2.2 SYNCHRONIZATION

We must have some way of communication between activities. We have the following primitives:

S y n t a x

```

wait      → {label:}WAIT  $\epsilon^+$  actname del
passive   → {label:}PASSIV  $\epsilon^+$  alist del
active    → {label:}ACTIVE  $\epsilon^+$  actname del
ifact     → {label:}ACTQ  $\epsilon^+$  actname, trlab del
ifterm    → {label:}TERMQ  $\epsilon^+$  actname, trlab del
iferror   → {label:}ERRQ  $\epsilon^+$  actname, trlab del
alock     → {label:}ALOCK  $\epsilon^+$  dataarea, trlab del
aunlock   → {label:}AUNLOC  $\epsilon^+$  dataarea del
alist     → actname {,actname}*

```

S e m a n t i c s

wait This will cause a wait until the activity actname is terminated. If actname is already terminated just go on, but if actname has never been created it is an error and will give an error interrupt.

passive Here the requesting activity will have to wait until it is activated from another activity. If a list is present it may be activated only from some activity in the list.

active This will cause the activity actname to be activated. If it is already active the activation will be queued until the activity is passivated.

ifact If actname is active go to trlab, else continue. Trlab must be in the scope of the requesting activity.

ifterm If actname is terminated (in error or not) go to trlab, else continue.

- 7.2.2** **iferror** The same as **TERMQ** but branch only if actname has terminated with an error status.
- alock** If the data area is assigned to another activity, then go to trlab else assign the area to the requesting activity.
- aunlock** Release the data area from requesting activity. The main activity must have performed a release of the data area before it may be assigned to another activity.

The first six primitives may be used for all sort of activities, but the last two may only be used for activities with data area. Notice that an activity created by **NEWACT** cannot have a data area!

7.2.3 ABSOLUTE PROGRAM LINKING

Sometimes it is convenient to create some absolute programs and then during execution link from one to another. In many systems the whole absolute program must be created as one big unit and it is impossible to communicate between absolute programs created at different times. The linking of absolute programs may be useful at interactive use of library programs.

O B M has the following linking facilities:

S y n t a x

link $\rightarrow$ {label:}LINK ϵ^+ absname del
up $\rightarrow$ {label:}UP $\epsilon^+ n$ del | {label:}UPPALL del
chain $\rightarrow$ {label:}CHAIN ϵ^+ absname del

S e m a n t i c s

link Here the control is transferred to the start address of absname. The return point of the calling program is remembered. Each link defines a new level. Linking of a program is not to be regarded as a new activity. The whole link may be regarded as one unit (activity). The program we are linking to is always a fresh copy of this. The calling program has to be saved and if necessary swapped.

- 7.2.3

chain

is similar to link, but we do not have a return point and the calling program need not to be saved. No return is possible.
- up

A return to the return point n levels up will be performed. If n is omitted, one is assumed. n is restricted to positive integers (zero not included). If n is greater than the number of levels it is an error. UPALL will cause a return to the first absname in the linking sequence.

7.3 SECONDARY MEMORY CONTROL

In OBMOS it is possible to assign a secondary memory area to a job. In OBM this memory area is called Secondary Logical Area (SLA), equivalent but ambiguous names in other systems are file or data set. This area has some properties defined in OBMOS (OBMOS command or default). The SLA is divided into items. One item is the smallest addressable unit from an OBM program. An item may be divided into logical subitems, a subitem is a character or a word. Word is here regarded as a compact bitstring of some defined length, character is a character of some length from some defined alphabet (it has a format).

In OBM there exist some standard SLA:s with standard names and properties. The names and some of the properties are given in the following table.

name	number of characters in one item	I / O restriction	F o r m a t
CARDS	80	I sequential	Standard OBM alpha- bet ISO646
PUNCH	80	O sequential	Standard OBM alpha- bet ISO646
PRINTER	120	O sequential	Standard OBM alpha- bet ISO646
DEMAND	72	I/O sequential	Standard OBM alpha- bet ISO646

7.3.1 I/O OPERATIONS

S y n t a x

inout → {label:}ioop[option] ε⁺device,buffer,number {,format} del
 ioop → WRITE|READ
 device → CARDS|PUNCH|PRINTER|DEMAND|sla|sla(position)
 sla → id
 position → integer|data
 buffer → data|data(ind)
 number → integer|adr
 format → type
 option → RFIN|RIM|LFIN|LIM|DFIN|DIM|ULFIN|ULIM

S e m a n t i c s

inout is the I/O statements in OBM. These statements are used for all transfer between the primary memory and secondary memories or I/O devices. The transfer is made between the device and a buffer in primary memory. The device may be text oriented, that is having characters as subitems or binary, that is having words as subitems. Text oriented devices have a format as a device property, if format is omitted in the statement the transfer is stright forward, but else the transferred characters are converted to format. The buffer must have an appropriate structure to receive the transferred characters.

For binary devices format has to be omitted and the transfer take place according to the structure of buffer.

There are two I/O statements, one for READ and one for WRITE. In order to perform any I/O the referenced device must be assigned to the activity. If there is an outstanding I/O operation for specified device a wait until finished will be performed. If an error occurs or if an end condition is satisfied, an I/O interrupt will occur. (See 9.2!)

- 7.3.1 device** The name of a device (sla) may have up to twelve significant characters. Name conventions are given in the OBMOS-CL.
- A SLA may be write or read protected. An I/O error interrupt occurs, if a prohibited action is demanded.
- If a SLA is referenced and it is locked by another activity (or another job), then a lockout interrupt occurs.
- buffer** The buffer must reside wholly within a pool or a string pool. For a binary device the pool must be built from words big enough to receive a word from the file, when reading and small enough when writing. For text oriented devices the same is true for characters, both ordinary pools and string pools may be used to contain the buffer.
- number** is used to specify the number of items to be transferred. If the transfer will cause an overflow of the pool, an out-of-pool interrupt occurs, (I/OOFL). The number must be ≥ 0 .
- format** The type must specify a string type with an alphabet. (See also string handling in 6.1!)
- option** In most computers it is possible to perform I/O-actions and processing in the CPU simultaneously. To take advantage of this it is possible to tell OBMOS if immediate return is desired or not. This and other modifications of I/O-action is specified by an option.

Option	I/O-restriction	SLA restriction	A c t i o n
RFIN	I/O		Return when I/O-action finished
RIM	I/O		Return immediately
LFIN LIM	I/O	Lock permitted	Set a lock, that locks out all other activities from this device. Then perform normal I/O-action with RFIN resp RIM
DFIN DIM	0	Dynamic	If end of device reached then perform necessary expansion (if max permitted size, then error interrupt). Perform write-action with RFIN or RIM!
UFIN ULIM	I/O		Perform I/O-action ended by a lock remove. Action performed as with RFIN or RIM!

7.3.1 position The position may only be specified for random access SLA. The position is the number of an item, the items are numbered from zero and upwards. A transfer will always update position to the next not transferred item, this is the only way of addressing items in a sequential file. When position is given in a transfer statement (random access SLA) it is the address of the first item to be transferred. An end of device will cause an interrupt. (See 9.2!)

7.3.2 I/O DEVICE CONTROL

It is also possible to control other device functions than data transfers. If a special device function is permitted or not depends on the SLA properties defined in OBMOS.

S y n t a x

devctrl	→	eof secexp secmove append lock unlock open close secsrch page devq ropen
devq	→	{label:}DEVQ ϵ^+ device,trlab del
eof	→	{label:}EOF ϵ^+ sla,del
secexp	→	{label:}EXPAND ϵ^+ sla,number del
secmove	→	{label:}MOVE ϵ^+ sla,number del
lock	→	{label:}ILOCK ϵ^+ sla del
unlock	→	{label:}IUNLOCK ϵ^+ sla del
open	→	{label:}OPEN ϵ^+ sla del
ropen	→	{label:}ROPEN ϵ^+ sla del
append	→	{label:}APPEND ϵ^+ sla del
close	→	{label:}CLOSE ϵ^+ sla del
secsrch	→	{label:}SECSRCH ϵ^+ sla,buffer,nkey,number del
nkey	→	integer adr
page	→	{label:}PAGE ϵ^+ action del
action	→	NEW LINE (integer)

- 7.3.2 devq** is a statement used to ask the system if a device is busy or not. If there is an outstanding (not finished) I/O operation for that device a branch to trlab is made, else the execution just continues with the next statement.
- eof** is a statement that causes an EOF item to be written on the sla. For a device like PRINTER this is ignored. When an EOF item is read, an EOF interrupt will occur, and position is updated to the first item after EOF.
- secexp** is a statement used to expand the size of a device. The number of items will be additionally assigned at the end of the SLA, if max limit is not reached. If max limit is reached an interrupt is generated.
- secmove** is used to move the position a number of items forward. If an EOF item is reached an EOF interrupt occurs. After this interrupt the device will be positioned just after the EOF item.
- lock** is used to perform the same as LFIN, LIM options, but without any data transfer.
- unlock** is used to perform the same as ULFIN, ULIM options, but without any data transfer.
- open** is used to assign sla to an activity for exclusive use. Any otherwise permitted action may be performed for this sla. The sla has to be catalogued and known to the system.
- ropen** is used to assign sla to an activity for reading only. Any number of activities may assign the same sla for read only action.
- close** is used to unassign sla for the calling activity. This is automatically done on activity termination, but it is good for system performance to close a sla as soon as possible.

7.3.2 `secsrch` is used to search a device for a special item. It is possible to search a sla forward a number of items. The SLA will be searched until a match is found between the nkey first sub items of the buffer, and the SLA item. After a successful match the SLA will be positioned at the found item. If no match is made a `nofind` interrupt occurs, and if an EOF item is found an EOF interrupt occurs.

`page` is used to position the PRINTER. `NEW` will give a new page (top) and `LINE` will position the printer to specified line on the page (only forward). The specified line must be within the limits of a page.

8. INTERRUPTS AND INTERRUPT SEGMENTS

In OBM the security is very important and it must be impossible to reach an undefined state. All errors must be detected and if not detected until runtime, they must cause an error interrupt. These interrupts are called error interrupts. Another form of interrupts is the I/O-interrupt that signals an unnormal condition to take care of, but this is not really an error.

For all interrupts standard interrupt routines are defined within the OBM system. It is possible to override the default action by a user defined interrupt routine. Those user defined actions may be any subroutine contained in some segment included in the program. It is also possible to build them from system routines. The registration of user interrupt routines must be done in an interrupt segment. At most one such a segment may be included in a program. The interrupt segment does not contain any executable statements, and no data.

S y n t a x

irptseg → label:IRPT del{irptreg}⁺END del

irptreg → errorirpt|ioirpt

S e m a n t i c s

irptseg The IRPT statement signals the start of the definition of an interrupt segment and defines label to be the name of such a segment.

irptreg This is a registration of an interrupt routine to take care of a specified interrupt.

END * This is the physical end of the interrupt segment.

8.1 ERROR INTERRUPTS

errorirpt → ERROR ϵ^+ errind,eirptdescr del

errind → id

eirptdescr → NOTHING|MSG|CALL(routine),{MSG|NOTHING}

routine → id

8.1 S e m a n t i c s

- errorirpt** The errorirpt statement registers an errorinterrupt routine for the error defined by errind. If NOTHING is specified the interrupt will be ignored, and if MSG is specified an error message will be printed and then the execution will continue as if no interrupt occurred. The third form of the eirptdescr will cause a normal subroutine call, within the interrupt routine.
- errorirpt** If NOTHING is specified the execution will continue after the return from the routine, and if MSG was specified the standard error message will also be printed, before the subroutine call is made.
If an error termination is wanted it is possible to perform an error termination within the subroutine (see operating system primitives). If another error interrupt of the same type (errind) will occur during execution of the user subroutine, the pointer back to the first interrupt sequence may be destroyed.
- errind** This is an index used to specify the type of interrupt. The non-standard interrupt routine is registered to take care of. These indices are described in the table in 8.3.
- routine** This is the name of a subroutine defined in one of the included executable segments. The routine must be a subroutine defined to have no parameters. The routine should be short and a return from this routine should be performed as soon as possible.

8.2 I/O INTERRUPTS

An I/O interrupt is not always an error, but rather a special case that should be returned to the user. The user is then responsible to perform an appropriate action. The user should define an I/O-interrupt routine.

8.2 S y n t a x

ioirpt $\rightarrow$ I/O ϵ^+ ioind, iodescr del

ioind $\rightarrow$ id

iodescr $\rightarrow$ CALL(routine), {MSG|NOTHING}|JUMP(trlab), {MSG|NOTHING}

S e m a n t i c s

ioirpt The ioirpt statement registers an I/O interrupt routine for specified interrupt (ioind). A CALL means that a subroutine is called in the same way as for error interrupts. The JUMP construction means that execution shall continue at address trlab and not where the interrupt occurred. MSG, NOTHING means message or no message. A new interrupt of the same type (ioind) during the called routine will lead to a logical error, the original return address is lost.

The trlab in the JUMP construction must be defined in an included executable segment as an externally referenceable transfer label.

ioind This is an index of an I/O interrupt as described in the following table. The I/O instructions and secondary memories SLA are described in 7.3.

8.3 INTERRUPT NAMES

TABLE OF ERROR INTERRUPTS

errind	default action MSG = message ERR = Error termination	Error description
OVERFL	MSG, ERR	Aritmetic overflow
DIVIDE	MSG, ERR	Divide fault
INDEX	MSG, ERR	Index out of range
PWIND	MSG, ERR	Partial word index out of range
DEF	MSG, ERR	Definition of stack/string outside of actual data area (pool)
STACK	MSG, ERR	Reference to element outside of stack/string max limits.
CASE	MSG, ERR	Label index outside of switch range.
SEG	MSG, ERR	Uncontrolled leaving of segment.
ACTS	MSG, ERR	Illegal start of activity (undefined prog).
LINK	MSG, ERR	Non-existing (registered) ABS referenced in link.
LOAD	MSG	Specified ABS cannot be loaded
WAIT	MSG	Wait for a non-existing activity.
ACTV	MSG	Activation of non-existing activity.
I/OERR	MSG, ERR	Unspecified I/O-error.
I/OOFL	MSG, ERR	Buffer overflow (outside Pool).
I/OMAX	MSG, ERR	Max limit for dynamic expansion reached

TABLE OF I/O INTERRUPTS

ioind	default action	interrupt description
EOD	MSG, ERR	Last item of SLA already transferred
EOF	MSG	An end of file item was read
I/OLOC	MSG	SLA locked by another activity
I/ONFD	MSG	A nofind condition during SEARCH

9. PROGRAM DESCRIPTION SEGMENT

The user can supply the linker with information about the program structure. The user is supposed to have some knowledge of the logical structure of the program, and the linker is supposed to know about the computer and its memory. As the user might be unaware of the computer and its memory size, he should divide his program into small segments and let the linker perform the final segmentation. The user should tell the linker about the logical structure of the program. This may be done in a program description segment, or by an OBMOS command. It is the responsibility of the OBMOS operating system to make an efficient segmentation for the actual computer. This means that a number of small segments may be merged into one segment. The information for OBMOS routines to make this efficiently is taken from:

1. The user supplied segmentation information.
2. Some cross reference tables constructed during translation.
3. Some library reference tables, contained in library.
4. The OBMOS knowledge of the actual computer.

Notice that the user supplied information is not a command to OBMOS, it is only a way of passing the user knowledge to OBMOS. This construction is used, because the user knows nothing of the computer and OBMOS does not know anything of the use of the program. But OBMOS knows the structure of the program from cross reference tables and perhaps from some analysis of the program.

S y n t a x

```

progdescr  → progstat mapstat * END del
progstat   → PROG {,irpt} + main {,unit} * del
mapstat    → label:MAP + unit {,unit} * del
unit       → primunit {/primunit} *
main       → seg
map        → id
irpt       → id

```


9. S e m a n t i c s

`progdescr` is the program description segment.

`progstat` is the first and defining statement in the program description. Notice that this segment cannot be named. By using the `irpt` as a specifier on the `progstat` the specified interrupt segment is used to define interrupt actions. This is the only way (besides on an OBMOS command) to specify user defined interrupt actions. The rest of the statement is used to describe the program structure. It is possible to use units defined by following map statements.

`mapstat` is used to give a name to a defined sub structure of the program. The label on this statement is used as the name of the substructure. The `progstat` and the `mapstat` use two binary infix operators/and, to describe the structure. The `/` has the highest priority, and the priority may be changed by parenthesis. The `/` operator is used to express what its two operands may start at the same location in memory (overlay segments). The `,` operator is used to describe a sequence of segments.

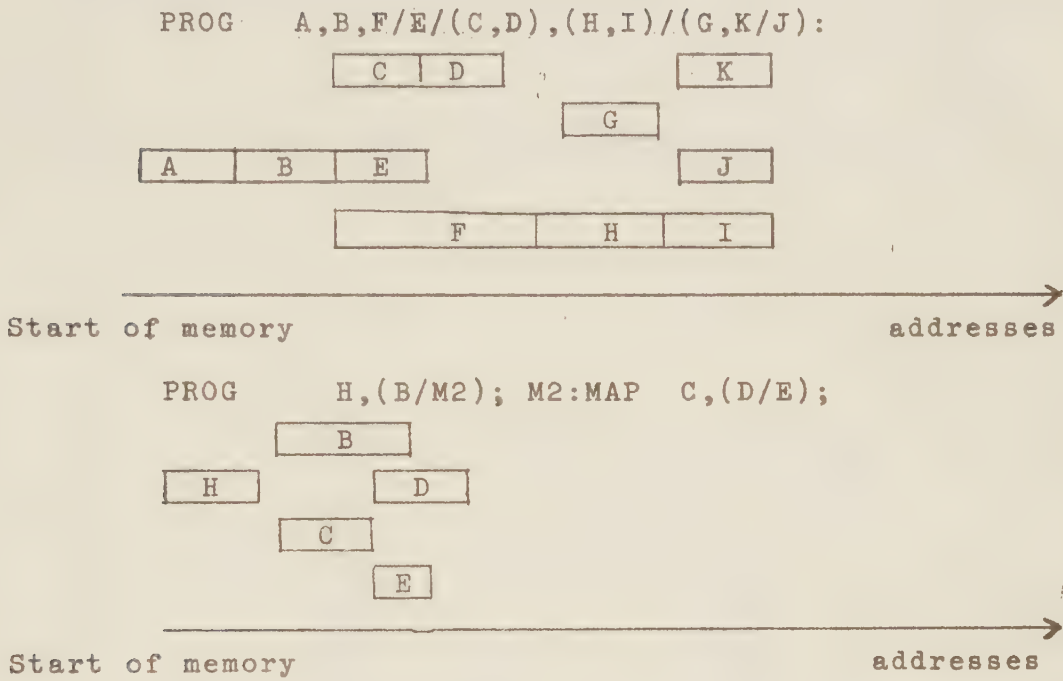
`main` is the name of a segment that contains a start address.

`map` is an identifier defined as a label on a `mapstat`.

9. Restrictions

The main segment must not be overlaid.

Examples:



Appendix A

The syntax of OBM.

This is a syntactic description with references to where in this report the syntax and semantics are defined. The semantic meaning of labels on a statement is also given.

Operator		Label	Reference	Comment
	$id \rightarrow letter \{ letter digit \}^*$ $integer \rightarrow 0 nonzero \{ digit \}^*$ $del \rightarrow \epsilon;$ $nonzero \rightarrow 1 2 3 4 5 6 7 8 9$ $digit \rightarrow 0 nonzero$ $letter \rightarrow A B C D E F G H I J K L M N O P Q R S T U V W X Y Z$ $adr \rightarrow id id(ind) id(ind, ind)$ $ind \rightarrow id integer -integer$ $data \rightarrow id$ $label \rightarrow id$ $trlab \rightarrow id$ $lab \rightarrow id$ $type \rightarrow id$			
PROGEND	$obmprog \rightarrow \{ progdescr \} \{ segment irptseg \}^+ PROGEND del$		2.2	six significant statement delimiter
SEGMENT	$segment \rightarrow label : SEGMENT \epsilon^+ \{ begin \} del$ $\{ exlab exref exsub datadescr \}^* \{ instruction \}^* END$			
END	$begin \rightarrow trlab$	seg	2.2 3.1	This is a program This is a normal segment
EXDEF	$exlab \rightarrow EXDEF \epsilon^+ trlab \{ , trlab \}^* del$	ignored		
EXREF	$exref \rightarrow EXREF \epsilon^+ id \{ , id \}^* del$	ignored		
SUBREF	$exsub \rightarrow label : SUBREF \epsilon^+ param \{ , param \}^* del$	sub		
	$datadescr \rightarrow typestat poolstat datastat alphastat pseudo stringstat common$			

Operator		Label	Reference.	Comment
TYPE	typestat $\rightarrow$ label:TYPE ϵ^+ tpar del {node} tpar $\rightarrow$ INTEGER,n{,store{,arithm}} REAL,exp,mant,base,sign,trunc REAL,ACTUAL STRING,ch,used,word,alpha NODE,pw,n BIT,n{,store} n $\rightarrow$ integer exp $\rightarrow$ integer mant $\rightarrow$ integer base $\rightarrow$ 2 4 8 16 10 ch $\rightarrow$ integer used $\rightarrow$ integer word $\rightarrow$ integer pw $\rightarrow$ integer store $\rightarrow$ N W S M E arithm $\rightarrow$ N S O C D sign $\rightarrow$ A B C trunc $\leftarrow$ N C R P alpha $\rightarrow$ id node $\rightarrow$ {wordstat} n_n wordstat $\rightarrow$ WORD{,E} ϵ^+ length{,length} * del length $\rightarrow$ integer integer*integer poolstat $\rightarrow$ label:POOL,type ϵ^+ {integer pool}{,{C R}}del pool $\rightarrow$ id	type	4.1	
WORD				
POOL		pool	4.2	

Computer	Label	Reference	Comment
STRING	stringstat	4.2	
	sopt		
	chars		
	bits		
COMMON	common		
ALPHA	alphastat		
	codelist		
	codeelement		
	code		
DATA	datastat		
	rel		
	value		
	int		
	float		
	stringval		
	character		
	struc		
	val		
	pseudo		
RADIX	radix		
	rad		
	checkop		
	arithrfault		
	recursion		
	instruction		
			not BNF description
			instruction part

Operator	Label	Reference	Comment
simpleinstruction			→ arithmetical logical branch stackstring keystat
arithmetical			→ transfer, compute, shift
transfer			→ move, abs, neg
move			→ {label:}= c^+ adr, adr{, type, type}del
abs			→ {label:}+=, type c^+ adr, adr del
neg			→ {label:}-=, type c^+ adr, adr del
compute			→ {label:}atmop, type c^+ adr, adr, adr del ;
REMAINDER			{label:}REMAINDER c^+ adr del
atmop			→ +, -, *, /
shift			→ {label:}shifto, type c^+ adr, adr, opnd del
opnd			→ adr, integer
shifto			→ /, -, //, +, //
logical			→ {label:}logop, type c^+ adr, adr, adr del
logop			→ AND, OR, XOR, MSKPC, MSKUPC
branch			→ case, casend, labass, goto, test
case			→ lavel:CASE c^+ trlab{, trlab}* del
casend			→ {label:}CASEND del
labass			→ {label:}LABEL c^+ labvar, trlab del
labvar			→ id
goto			→ {label:}GO c^+ {swlab(ind) trlab labvar}{, retlab} del
retlab			→ trlab
swlab			→ id
test			→ zerotest, comptest, memtest
zerotest			→ {label:}ztop, type c^+ adr, trlab{, trlab}del

5.3.1

trlab
trlab
trlab
trlab
trlab

trlab

trlab 5.3.2

5.4

swlab 5.4.2

trlab 5.4.3

trlab 5.4.2

trlab 5.4.3

5.4.4

trlab

IF=0, IF<0
IF>0, IF><0

Operator	Label	Reference	Comment
IF=, IF< IF>, IF<< IFMEMB	trlab		ztop → IF=0 IF<0 IF>0 IF><0 comptest → {label:}top,type c ⁺ adr,adr,trlabf,trlab} del top → IF IF< IF> IF>< membtest → {label:}IFMEMB,type c ⁺ adr,adr,adr,trlabf,trlab} del stacktradr → strdef stkhd strdef stkhd1
DEFINE	str, trlab	6.1	strdef → label:DEFINE,STRING c ⁺ string,size,type,finish del string → id size → integer adr finish → trlab
GETCH	trlab	6.1	strhdl → get chmove setptr pack unpack get → {label:}GETCH c ⁺ str,adr del
CMOVE	trlab	6.1.1	chmove → {label:}CMOVE{,CONV} c ⁺ str(ind),str(ind),num del str → id num → integer adr
GETPTR	trlab		getptr → {label:}GETPTR c ⁺ {str stk},adr del
SETPTR	trlab		setptr → {label:}SETPTR c ⁺ {str stk},adr del
PACK	trlab		pack → {label:}PACK c ⁺ str(ind),num,adr del
UNPACK	trlab		unpack → {label:}UNPACK c ⁺ str(ind),num,adr del
DEFINE	stk trlab	6.1.2	stkdef → label:DEFINE,STACK c ⁺ stkbase,size,underflow,overflow del stkbase → data data(ind) underflow → trlab overflow → trlab stk → id stkhd1 → stack pop top getptr setptr

Operator	Label	Reference	Comment
STACK			stack $\rightarrow$ {label:}STACK e^+ stk,adr del
POP	trlab		pop $\rightarrow$ {label:}POP e^+ stk,adr del
TOP	trlab		top $\rightarrow$ {label:}TOP e^+ stk,adr del
REPEAT	trlab	5.2	lopp $\rightarrow$ {label:}REPEAT e^+ index,start,step,fin del
SECIND			{SECIND e^+ secind(index,numb)}{,secind(index,numb)} * del
REPEND			{exit,instruction} $^+$ REPEND e^+ {adr} del ;
	trlab		{label:}REPEAT.L e^+ times del{exit/instruction} $^+$ REPEND del
			index $\rightarrow$ id
			start $\rightarrow$ integer adr
			step $\rightarrow$ integer adr
			fin $\rightarrow$ integer adr
			secind $\rightarrow$ id
			numb $\rightarrow$ integer
LEAVE	trlab		exit $\rightarrow$ {label:}LEAVE e^+ adr del
			times $\rightarrow$ integer adr
			subroutine $\rightarrow$ nonrec rec foreign
NONREC	sub	5.3	nonrec $\rightarrow$ label:NONREC e^+ {paramlist} del
			{simpleinstruction loop return} $^+$
ROUTEND			ROUTEND del
REC	sub		rec $\rightarrow$ label:REC e^+ {paramlist} del
LOCAL			{LOCAL e^+ loclist del} * {simpleinstruction loop return} $^+$
			ROUTEND del
FSUB	sub		foreign $\rightarrow$ label:FSUB e^+ fsubdescr del
			paramlist $\rightarrow$ param{,param} *

Operator	Label, Reference	Comment
RETURN	trlab	<pre> param → data(lng,ptype) lng → type'integer ptype → 0'1 2 return → {label:}RETURN del loclist → id{,id}* fsubdescr → (wn,wl){,(wn,wl)}* wn → integer wl → integer call → {label:}CALL,sub {s⁺ actparlist} del sub → id actparlist → {integer adr {,{integer adr}}* keystat → label:KEY,type s⁺ data{(ind)} {,mask} del mask → adr integer integer* search → {label:}srchop s⁺ adr,data{(ind)} ,key,number,nofind del srchop → SRCH< SRCH< SRCH> SRCH>< SRCHZ key → id number → adr integer nofind → trlab system → exprim inout devctrl exprim → exec fork nact aexit err abort wait passive active ifact ifterm iferror alock aunlock chain link up exec → {label:}EXEC s⁺ absname del absname → id fork → {label:}FORK s⁺ actname,absname{,dataarea} del actname → id dataarea → id </pre>
CALL	trlab	
KEY	key	6.4
SRCH= SRCH>	trlab	
SRCH< SRCH><		
SRCHZ		
		7.1
EXEC	trlab	7.2.1
FORK	trlab	

Operator	Label	Reference	Comment
NEWACT	nact → {label:}NEWACT ε ⁺ actname, trlab del		trlab
EXIT	aexit → {label:}EXIT del		trlab
ERROR	err → {label:}ERROR del		trlab
ABORT	abort → {label:}ABORT del		trlab
WAIT	wait → {label:}WAIT ε ⁺ actname del	7.2.2	trlab
PASSIV	passive → {label:}PASSIV {ε ⁺ alist} del alist → actname{,actname} [*]		trlab
ACTIVE	active → {label:}ACTIVE ε ⁺ actname del		trlab
ACTQ	ifact → {label:}ACTQ ε ⁺ actname, trlab del		trlab
TERMQ	ifterm → {label:}TERMQ ε ⁺ actname, trlab del		trlab
ERRQ	iferror → {label:}ERRQ ε ⁺ actname, trlab del		trlab
ALOCK	alock → {label:}ALOCK ε ⁺ dataarea, trlab		trlab
AUNLOC	aunlock → {label:}AUNLOC ε ⁺ dataarea del	7.2.3	trlab
CHAIN	chain → {label:}CHAIN ε ⁺ absname del		trlab
LINK	link → {label:}LINK ε ⁺ absname del		trlab
UP	up → {label:}UP {ε ⁺ n}del {label:}UPALL del		trlab
UPALL	inout → {label:}ioop{,option} ε ⁺ device,buffer,number{,format} del		trlab
WRITE READ	ioop → WRITE READ		trlab
	option → RFIN RIM LFIN LIM DFIN DIM ULFIN ULIM		trlab
	device → CARDS PUNCH PRINTER DEMAND sla sla(position)		trlab
	sla → id		trlab
	position → integer data		trlab
	buffer → data data(ind)		trlab
	format → type		trlab

	Reference	
devctrl → eof seexp secmove append lock unlock ropen open close secsrch page devq	7.3.2	
eof → {label:}EOF ε ⁺ sla del		trlab
seexp → {label:}EXPAND ε ⁺ sla,number del		trlab
secmove → {label:}MOVE ε ⁺ sla,number del		trlab
append → {label:}APPEND ε ⁺ sla del		trlab
lock → {label:}LOCK ε ⁺ sla del		trlab
unlock → {label:}UNLOCK ε ⁺ sla del		trlab
ropen → {label:}ROPEN ε ⁺ sla del		trlab
open → {label:}OPEN ε ⁺ sla del		trlab
close → {label:}CLOSE ε ⁺ sla del		trlab
secsrch → {label:}SEARCH ε ⁺ sla,differ,nkey,umber del		trlab
nkey → integer adr		
page → {label:}PAGE ε ⁺ action del		trlab
action → NEW LINE(integer)		
devq → label: DEVQ ε ⁺ device,trlab del		trlab
irptseg → label:IRPT del irptreg ⁺ END del	8	irpt
irptreg → errorirpt ioirpt		
errorirpt → ERROR ε ⁺ errind,eirptdescr del	8.1	
errind → id		
eirptdescr → NOTHING MSG CALL(routine),{MSG NOTHING}		
routine → id		
ioirpt → I/O ε ⁺ ioind,iodescr del	8.2	
ioind → id		
iodescr → CALL(routine),{MSG NOTHING} JUMP(trlab),{MSG NOTHING}		

Operator	Label	Reference	Comment
PROG		9	
MAP	map		
EMPTY	trlab	5.2	

```

progdescr → progstat{mapstat}* END del
progstat → PROG{,irpt}c+ main{,unit}* del
mapstat → label:MAP c+ unit{,unit}* del
unit → primunit{/primunit}
primunit → mapseg|(unit{,unit}*)
main → seg
map → id
irpt → id
empty → {label:}EMPTY del

```


APPENDIX B

RESTRICTIONS

These restrictions are valid if not the semantic definition implies stronger restrictions.

Operator / statement	Restrictions
EXDEF	
EXREF	
SUBREF	
TYPE	INTEGER $1 < n < 256$, if decimal arithmetic $7 < n < 256$ REAL $2 < \text{exp} < 65$, $1 < \text{mant} < 253$ $\text{exp} + \text{mant} < 256$ STRING $1 < \text{used} \leq \text{ch} < 9$, $0 < \text{word} * \text{ch} < 256$ BIT $0 < n < 256$ NODE $0 < \text{length}$ $\text{length} < 256$ word, E $0 < \text{pw} <$
ALPHA	$1 < \text{bits} < 9$, $0 \leq \text{code} < 128$
POOL	$\text{size of pool in words} \leq 2^{16}$ $\text{size of pool in bits} \leq 2^{20}$
STRING	$1 < \text{bits} < 9$, $0 < \text{chars} < 2^{16}$
RECLIM	$\text{stacksize} < 2^{16}$
opnd	$\text{opnd given as integer} < 2^{16}$
DEFINE	$\text{size} < 2^{16}$
CMOVE	$\text{num} < 2^{16}$
PACK	
UNPACK	
REPEAT	$\text{start}, \text{step}, \text{fin as integers} < 2^{16}$, $\text{times} < 2^{16}$
SECIND	$\text{numb} < 2^{16}$
FSUB	$\text{wn} < 2^6$, $\text{wl} < 2^8$
loop	$\text{position} < 2^{32}$

APPENDIX C

EXAMPLE OF A PROGRAM IN ALGOL AND OBM, COMPUTING THE
PRIMES LESS THAN 100

```

1  BEGIN INTEGER I,K,FIRST;
2      EXTERNAL PROCEDURE PRINT ;
3      BOOLEAN ARRAY PRIME(2:100);
4      FOR I:=2 STEP 1 UNTIL 100 DO PRIME(I):=TRUE;
5      FOR I:=2 STEP 1 UNTIL 10 DO
6          IF PRIME(I) THEN
7              BEGIN FIRST:=2*I;
8                  FOR K:=FIRST STEP I UNTIL 100 DO PRIME(K):=FALSE;
9              END;
10     FOR I:=2 STEP 1 UNTIL 100 DO
11         IF PRIME(I) THEN PRINT(3,1,I);
12 END

```

```

1  MAIN:SEGMENT START;
2      BOL:TYPE INTEGER,1; H:TYPE INTEGER,16;
3      LOG:POOL,BOL 10000; NUM:POOL,H 2;
4      FALSE:DATA,BOL LOG,0,1; PRIME:DATA LOG,1;
5      FIRST:DATA NUM,0; TWO:DATA,H NUM,1,2;
6      OUT:SUBREF A(16,0);
7      START:REPEAT I,2,1,10 ;
8          IF><0,BOL PRIME(I),LO;
9          *,H FIRST,I,TWO;
10         REPEAT K,FIRST,I,100 ;
11             = PRIME(K),FALSE;
12         REPEND;
13     LO:REPEND;
14     REPEAT I,2,1,100 ;
15         IF><0,BOL PRIME(I),L1; CALL,OUT,I;
16     L1:REPEND;
17 END;
18     PROGEND;

```